TURING 图灵电子与电气工程丛书

PEARSON Prentice Hall

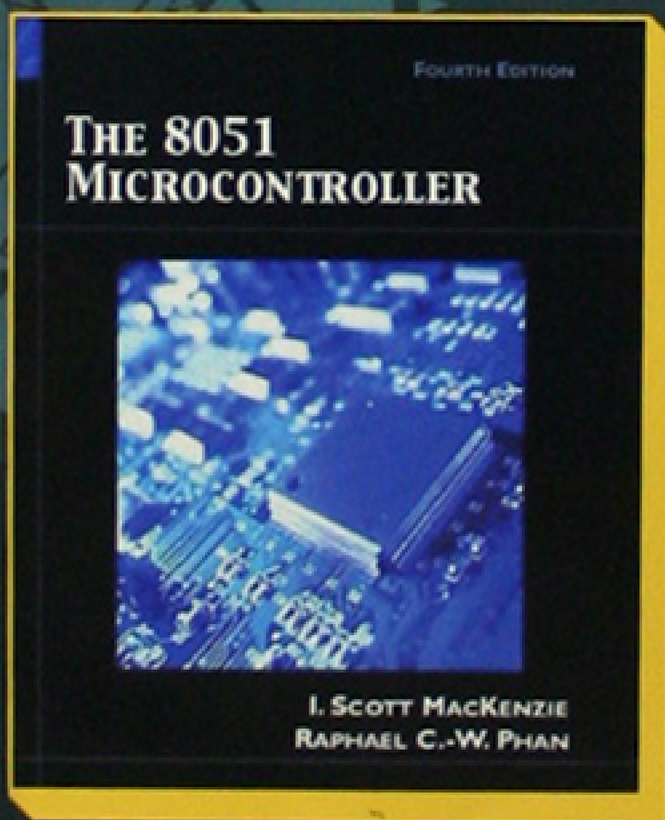
8051微控制器

(第4版)

The 8051 Microcontroller

Fourth Edition

[加] I. Scott MacKenzie
[马来西亚] Raphael C.-W. Phan 著
张瑞峰 等译
李 犇 审校



人民邮电出版社
POSTS & TELECOM PRESS

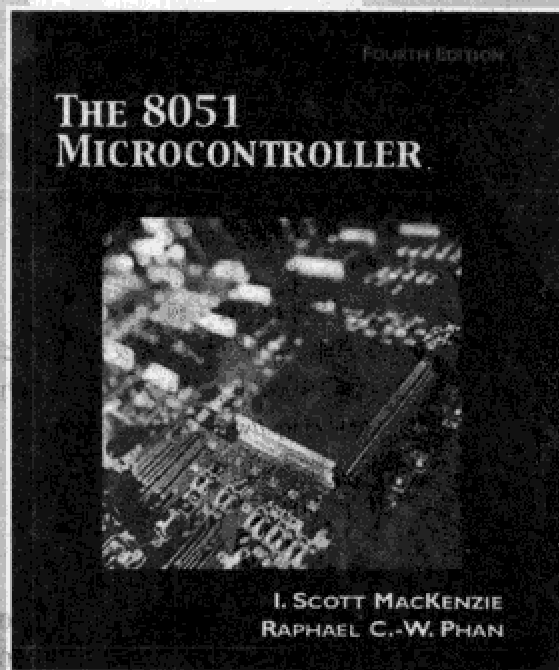
TURING 图灵电子与电气工程丛书

8051微控制器

（第4版）

The 8051 Microcontroller
Fourth Edition

[加] I. Scott MacKenzie
[马来西亚] Raphael C.-W. Phan 著
张瑞峰 等译
李 锵 审校



人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

8051微控制器：第4版/（加）麦肯齐（MacKenzie, I.S.）（马来西亚）法恩（Phan, R. C. -W.）著；张瑞峰等译. —北京：人民邮电出版社，2008.7

（图灵电子与电气工程丛书）

书名原文：The 8051 Microcontroller, Fourth Edition

ISBN 978-7-115-17959-3

I. 8… II. ①麦… ②法… ③张… III. 微控制器 IV. TP332.3

中国版本图书馆CIP数据核字（2008）第051838号

内容提要

本书介绍以8051为代表的MCS-51系列微控制器的硬件和软件方面的基本知识和特性，着重描述其硬件体系结构和软件编程问题。同传统的微控制器书籍相比，本书更注重技术上的实现细节，着眼于教会读者如何解决具体的工程问题。在软件设计方面，同时给出了汇编程序和8051 C语言程序，讲解了C语言在复杂8051项目中的优势。本书在讲解重要的基本概念和方法时都给出了例题，便于读者掌握理解。

本书可作为高等院校相关专业教材，也适合从事单片机和嵌入式系统开发的工程技术人员。

图灵电子与电气工程丛书

8051微控制器（第4版）

◆ 著　［加］I. Scott MacKenzie, ［马来西亚］Raphael C.-W.Phan
译　张瑞峰　等
审　校　李　锵
责任编辑　朱　巍

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京铭成印刷有限公司印刷
新华书店总店北京发行所经销

◆ 开本：700×1000　1/16
印张：22
字数：635千字　2008年7月第1版
印数：1-4 000册　2008年7月北京第1次印刷

著作权合同登记号　图字：01-2006-5677号

ISBN 978-7-115-17959-3/TN

定价：49.00元

读者服务热线：（010）88593802　印装质量热线：（010）67129223

反盗版热线：（010）67171154

译 者 序

在市场需求和技术进步的双重驱动下，微控制器领域的新产品层出不穷，但应用最广、技术最成熟的还是最早由Intel公司开发的MCS-51系列微控制器，目前该系列微控制器仍然占据着非常大的市场份额。众多半导体公司（如Atmel、Microchip、Motorola、Philips、Cypress等）获得了Intel公司的授权，不断推出功能更强大、集成度更高、各具特色的8位微控制器产品，因此基于8051的微控制器教材也一直长盛不衰。本书就是在这一背景下推出的，而且一版再版，这已经是第4版了，由此可见其受欢迎的程度。作者在不断改进、精炼旧有内容的同时，又结合当前微控制器领域的发展前沿，增加了很多新内容。与传统的微控制器书籍相比，本书具有如下特点。

- 注重细节。比如在讨论布尔指令时，将采用8051微控制器和数字逻辑电路分别实现与运算所花费的时间做了比较，后者要比前者快1000倍；这反映了微控制器在实时性要求较高的场合的局限性。作者没有一味强调微控制器的优点，而是客观地指出其应用受限之处。另外，在讨论如何通过定时器产生精确延时问题时，提出了两种技术方案，一种是通过选择合适频率的晶振来修正重载值计算产生的舍入误差，另一种是通过精心计算调整计数初值的方式来补偿指令执行所消耗的时间。书中对很多此类在时序和控制逻辑方面的技术细节都阐述得透彻明了，这种求真严谨的作风非常值得学习。
- 工程观念很强。本书并没有过于追求对基本理论和硬件电路细节的描述，而是着眼于教会读者在面对一个具体的工程问题时，如何动手解决问题。即首先得到市场或客户所提出的技术指标要求，然后启动产品的开发流程（包括整体方案规划、硬件和软件规范描述），最后是整个系统的整合和验证。

在软件设计方面，第11章和第12章将汇编程序和8051 C语言程序对照给出，使读者很容易发现二者的联系、区别和优缺点，克服了将汇编和C分立讲述的弊端（有些教材只讲汇编或者只讲C）。本书用大量篇幅讨论了汇编和链接操作过程的细节，有利于读者掌握由汇编代码转换成机器码的来龙去脉，并详细介绍了ASM51的汇编伪指令和汇编控制项。本书强调结构化程序设计的重要性，并对伪码和流程图两种设计方法进行了精彩的阐述。先写伪码或画流程图，然后再依此编写源代码，并附加详尽的注释。这是非常好的编程习惯。

本书作者善用举例说明的方法来论述问题，在讲解重要的基本概念和方法时都给出了恰当的例题，并在解答问题之后展开富有创造性和开拓性的讨论，对读者很有帮助。

另外，与一般的微控制器教程不同，端口和存储器扩展的内容被整合到具体的例子中讲述（第11章），形象具体，便于读者掌握。第13章提供了7个具有综合性、代表性的学生练习项目，并给出了项目描述、硬件列表、系统设计和软件设计（伪码）的大致轮廓，如果读者将这些项目付诸实际（设计原理图、PCB版图，购买元器件，采用C或汇编编程、调试），相信利用8051解决实际工程问题的能力一定会大有提高。

每章后的大量习题进一步充实了本书的内容，同时也可以帮助读者更好地理解基本理论。本书共有11个附录（A~K），内容全面详尽，对初学者和工程技术人员而言都是难得的技术手册。

该书既可以作为高等院校电子、信息、通信、计算机等专业本科生微处理器课程的教材，又可以供相关工程技术人员参考。

本书由张瑞峰主译，李锵审校。参加本书翻译和初校工作的还有詹敏晶、张为、张培源、关欣、刘艳艳、杨爱萍、肖志涛、汪剑鸣、王昕、郭琦、赖焰根、关剑、刘航、李晓、沈运强、马杰等。在此，谨对所有为本书的出版提供了帮助的人们表示诚挚的谢意！

由于译审者水平有限，加之时间仓促，译文中难免有不妥乃至错误之处，敬请读者不吝指正。

前　　言

本书介绍了以8051为代表的MCS-51系列微控制器的硬件和软件方面的基本知识和特性，适用于高等院校电子技术或计算机工程专业师生以及对微控制器技术感兴趣的工程技术人员。

作者在本书编写过程中不断了解读者有关8051微控制器的信息需求，并对书中内容加以精炼。本书第1版内容基于为计算机工程专业的本科生第5学期开设的一门微控制器课程。如第11章所述，在该课程中，学生将动手制作一个8051单板机。在第6学期的"项目"课程中，学生将以该单板机为目标系统，进行基于8051微控制器产品开发方面的训练，包括方案设计、软硬件实现和编写技术文档等工作。

同其他微控制器一样，8051自身的功能很强大，本书将着重描述其硬件体系结构和软件编程问题，而忽略电气方面的一些细节。有关软件的一些主题，请参考Intel公司的汇编器ASM51和链接/定位器RL51的说明文档。

本书在以前版本基础上增加了4章，主要是采用8051 C语言编程取代旧版本中汇编语言编程所产生的一些新的变化。C语言容易实现结构化编程思想，而且比较适合于代码量大并且复杂的8051项目。

书中的所有例题都做了相应的注释，便于老师和学生理解。例题以陈述问题、给出答案、深入探讨问题和解决方案的方式给出。这是一种详细解释和阐述问题的方式，使读者可以从多个角度来理解例题。

由于很难将一些讨论主题合理排列，所以微处理器和微控制器的课程一般比其他课程（如数字系统）要难讲一些，当然这只是我们的一家之言。在给出学生第1个例题时实际上是做了如下假设的，即他们已经掌握了CPU编程模式和寻址模式、地址和地址对应空间存储的内容之间的区别等基础知识。基于上述原因，教师在基于本书讲授该课程时可以灵活处理，而不必严格按照本书的章节顺序授课。但本书的第1章是一个很好的切入点，不仅提供了微控制器的一般性介绍，而且强调了微处理器和微控制器二者之间的差别。

第2章主要介绍了8051微控制器的硬件体系结构和MCS-51系列产品，并以短小的指令序列形式给出了一些简明例题。在这一点上，学生需要掌握一些关于汇编指令的基本知识，同时可以参考第3章和第7章以及附录A和附录C来达到此目的。附录A特别实用，因为其中有一张关于8051指令集的完整表格。

第3章的主要内容是8051指令集。首先介绍8051的各种寻址模式，然后将指令（数据传送、分支等）分门别类地逐一介绍，并用许多简短的例子来辅助说明各种寻址模式和指令。

第4章~第6章主要介绍8051的片上功能。首先介绍定时器，接着介绍串行端口（需要采用定时器作为波特率发生器），最后介绍中断。这几章中的例题比之前的更长、更复杂。教师不要急于讲授这3章的内容，因为学生在学习这些内容之前，对8051的硬件体系结构和指令集应该有扎实全面的理解。

第7章的一些主题在前面6章中已经涉及了（如有必要）。但是，在该章中还有一些有利于培养学生承担大规模项目的重要内容。一些比较高级的主题，如汇编时的表达式求值、模块化编程、链接/定位和宏编程等，对许多学生而言都是很大的挑战。从这一点来看，需要非常强调动手经验的重要性。应该鼓励学生亲手将本章的例题输入到计算机里，通过ASM51和RL51等汇编器查看输出结果和出错信息以及目标文件到十六进制文件的转换过程（OH）。

第8章讲述8051 C语言编程的基本知识，主要突出两点：高级语言和汇编语言之间的区别，针对计算机系统的传统C语言和针对8051等嵌入式微控制器的C语言之间的区别。

第9章和第10章讨论了有关编程方法、风格和开发环境等高级主题。这两章范围更广泛，概念性更强。在专业开发领域，这些内容显得尤为重要。

第11章介绍了一些与选定的硬件和所支持软件相结合的设计示例。软件加有详尽的注释，是这些例题的重点所在。在本版中还增加了一些接口示例：液晶显示模块（LCD）、8255、RS-232串行接口、CENTRONICS并行接口、传感器、继电器和步进电机。其中，SBC-51单板机可算作8051微控制器课程的基础，还包括一个小的监控程序（如附录G所示），它已经具备了使8051微控制器“启动运行”的基本条件。开发环境还需要一台主机，而且，主机还同时兼作控制SBC-51的哑终端（在程序下载到8051上执行之后）。

很多计算机工程专业的学生在修读作者所授的8051微控制器课程期间，动手布线实现了SBC单板机的原型。同时也要感谢学习微处理器基础、微处理器应用和嵌入式微控制器课程的学生们，是他们热心地完成了课上布置的基于8051的项目设计任务。

第12章也是在第4版中新增的，其中的设计和接口示例完全和第11章相同，只是以C语言代替汇编语言来完成这些项目。

第13章给出了供学生练习的一些比较高级的8051项目示例，重点讨论如何根据要求设计项目的基本方案以及在设计程序时伪码（需要在编写实际代码之前完成）的重要性。

第14章简要介绍了8051的衍生产品，这些产品增强了原8051的某些功能，如提

高了运行速度、增加了存储器大小、增加了内嵌的外围电路、增强了网络性能和安全机制。

还值得一提的是，在第4版中也增加了关于智能卡和数据安全方面的内容，具体内容见第12章~第14章及附录J。之所以将这部分内容列入本书，是因为目前在智能卡中利用8位微控制器（如8051）来运行安全性软件从而保护机密信息已经越来越普遍。

本书引用了很多Intel公司有关MCS-51的技术文档，而且附录C包含了所有8051指令的定义，附录E包含了8051数据表。在此向Intel公司深表谢意。

本版中的所有8051 C程序都在Keil公司的μVision2集成开发环境中编译、调试、测试通过，该软件可以从http://www.keil.com网址下载。感谢下面这些朋友对本书的审阅、评价、批评和建议：内华达大学的Dwight Egbert，加州理工州立大学的Marty Kaliski，俄勒冈技术学院的Claude Kansaku，圣达菲社区学院的Ron Tinkham。Raphael还要感谢他的爱人Grace，感谢她的理解和支持。在编写本书期间，Grace牺牲了很多个夜晚、周末和休假来维持公司的正常运转。事实上，如果没有她的奉献就无法完成本书。因此，将本书献给她聊表谢意。

I.Scott MacKenzie
Raphael C.-W. Phan



目　　录

第1章 微控制器简介

1.1 引言

虽然计算机问世刚刚几十个年头，但就如同当年的电话、汽车和电视机的出现一样，深刻地改变了人类的生活。在当今社会，无论是计算机程序员还是每月通过邮件收到大型计算机系统打印的银行信用卡账单的普通消费者，每个人都会感觉到计算机的存在。在我们的观念里，计算机已经被定义为具有无穷无尽能力的执行数字操作的“数据处理器”。

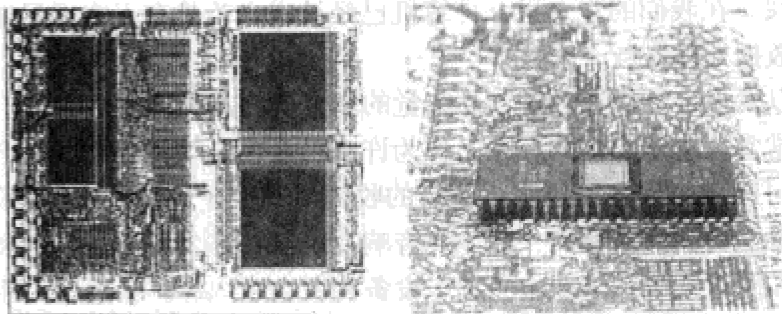
我们可以发现计算机在各种不易察觉的场合安静高效地完成着人们赋予的工作任务，甚至常常被忽略它们的存在。作为许多工业、自动化和消费类产品的核心部件，计算机可以应用到各种场合：超市的收银机和电子秤；家中的烤箱、洗衣机、闹钟、温控器、玩具、录像机、立体声音响及乐器；办公室的打字机和复印机；汽车的仪表盘和点火系统；工业中的钻床设备和照排系统等。在这些应用中，计算机主要通过和“现实世界”的接口完成“控制”功能，如设备的开启和关闭以及工作状态的监测。在上述领域中还经常用到**微控制器**（与微型计算机或微处理器相对应）。

很难想象如果现在的电子类产品中没有微处理器将会是怎样一番景象；尽管这个单芯片奇迹出现刚刚30多个年头。1971年，Intel公司推出了第一款成功的微处理器8080。此后不久，摩托罗拉、RCA、MOS Technology、Zilog等公司纷纷推出了类似的产品：6800、1801、6502和Z80。虽然在缺乏外围辅助元件的情况下，这些集成电路的功能无法发挥出来，但是当被嵌入到单板机（SBC）里面后，它们将会成为有用产品的核心部件，通过它们可以了解微控制器，进而用微控制器进行设计。较著名的单板机有摩托罗拉的D2、MOS Technology 的KIM-1、Intel的SDK-85，这些单板机一经推出便迅速进入了大学、科研院所及电子公司的设计实验室。

微控制器是同微处理器相似的器件。1976年，Intel公司推出了8748，这是MCS-48™系列的第一款产品。该芯片集成了17 000多个晶体管，包含一个CPU、1KB的EPROM、64B的RAM、27个I/O引脚和8位的定时器。该芯片及后来推出的其他MCS-48™系列芯片迅速成为面向控制应用的工业标准。起初这些器件被大量用于洗衣机、交通信号灯控制器，替换机电元件。现在微控制器的应用领域已经拓展到汽车、工业设备、消费类电子娱乐产品以及计算机的外围设备。（如IBM PC机的

键盘内部就含有微控制器芯片，这是一个元件最少的设计范例。）

1980年Intel公司推出了其MCS-51™系列微控制器第一款产品8051，由此，微控制器在功耗、尺寸和复杂性上都提升了一个数量级。与8048相比，该器件包括60 000多个晶体管、4KB的ROM、128B的RAM、32个I/O引线、1个串行端口以及2个16位定时器。其单芯片上的电路数量具有划时代意义（如图1-1所示）。此后MCS-51™系列产品不断壮大，时至今日，微控制器芯片的规模已经达到第一款51芯片的两倍水平了。作为全球第二款MCS-51™微控制器芯片的创造者，西门子公司开发出了SAB80515，它具有增强型8051内核、68个引脚、6个8位的通用I/O端口、13个中断源、8输入通道8位A/D转换器。第14章还将讨论一些增强型8051衍生产品。目前8051系列产品已经成为8位微控制器中应用最广泛、功能最强大的一大种类，并且在未来几年内其领军位置不会改变。



(a) 8051　　(b) 带有片上EPROM的8751

图1-1　8051微控制器（得到Intel公司惠允）

本书的研究对象是MCS-51™系列微控制器。在后续的章节中将介绍MCS-51™系列微控制器的硬件和软件体系结构，并通过大量设计实例说明如何使用该系列微控制器、以最少数目的外加元件完成相应的电子设计。

2

下面将通过对计算机体系结构的简要讲解，介绍一些该领域广泛使用（通常也容易混淆）的缩略语和专业词汇。一些大公司的偏见和不同设计者的不同理解造成了很多术语的定义含糊不清或者相互重叠，本书对此的处理原则是重实践应用、轻学术理论，以最通俗的语言对每个术语给出直接明了的解释。

1.2　术语

计算机的定义首先要包含两个关键特征：(1) 具有无人干预条件下通过程序处理数据的能力，(2) 具备存取数据的能力。更一般地讲，**计算机系统**也包括用于人机通信的**外围设备**，还包括处理数据的**程序**。其中的设备称为**硬件**，程序称为**软件**。下面结合图1-2开始介绍计算机的硬件。

显而易见，图1-2是一个抽象的硬件体系结构，用来代表所有类型的计算机。

如图所示，计算机系统包含一个**中央处理器**（CPU），CPU通过**地址总线**、**数据总线**和**控制总线**连接**随机访问存储器**（RAM）和**只读存储器**（ROM）。**接口电路**是系统总线和**外围设备**的联系枢纽。下面详细讨论上述硬件单元。

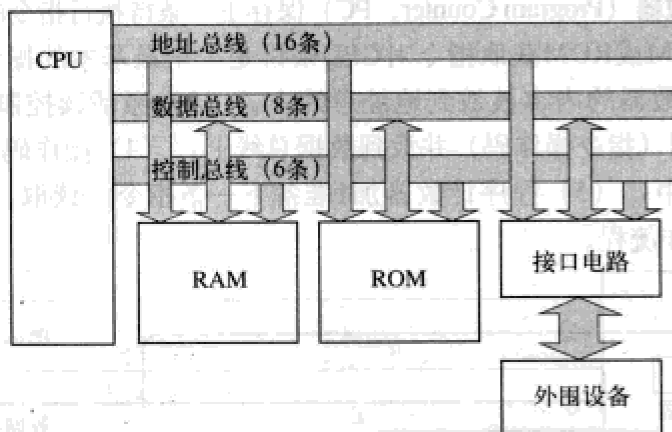


图1-2 微型计算机系统框图

1.3 中央处理器

CPU作为计算机系统的“大脑”，负责管理系统的所有活动并处理所有数据操作。CPU其实并不神秘，因为它只是集成了很多逻辑电路，周而复始地做两件事情：获取指令和执行指令。CPU具有理解和执行基于二进制码的指令的能力，每条指令表示一个简单操作。这些指令通常用于执行算术（加减乘除）、逻辑（与或非等）、数据传送或者分支转移等运算，且由一组称为**指令集**的二进制码来表示。

图1-3是CPU内部结构的简单示意图，包括一组用于临时存储信息的**寄存器**、一个

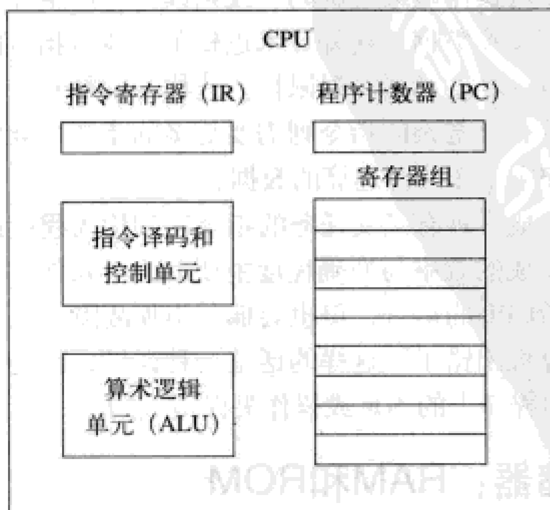


图1-3 中央处理器（CPU）

用于完成信息操作的**算术逻辑单元**（Arithmetic and Logic Unit，ALU）、一个**指令译码和控制单元**（它决定需要执行的操作，将指令译码为完成操作所需要执行的动作）以及两个寄存器。**指令寄存器**（Instruction Register，IR）用于保存当前执行的指令的二进制码，**程序计数器**（Program Counter，PC）保存下一条待执行指令的存储器地址。

从系统的RAM或ROM获取指令对CPU来讲是一个最基本的操作，包括下列步骤：（1）程序计数器的内容被放到地址总线上；（2）激活读控制信号；（3）从RAM中读取数据（指令操作码）并放到数据总线上；（4）操作码被锁存到CPU的内部指令寄存器中；（5）程序计数器加1准备下一条指令的读取。图1-4描述了指令读取过程的操作流程。

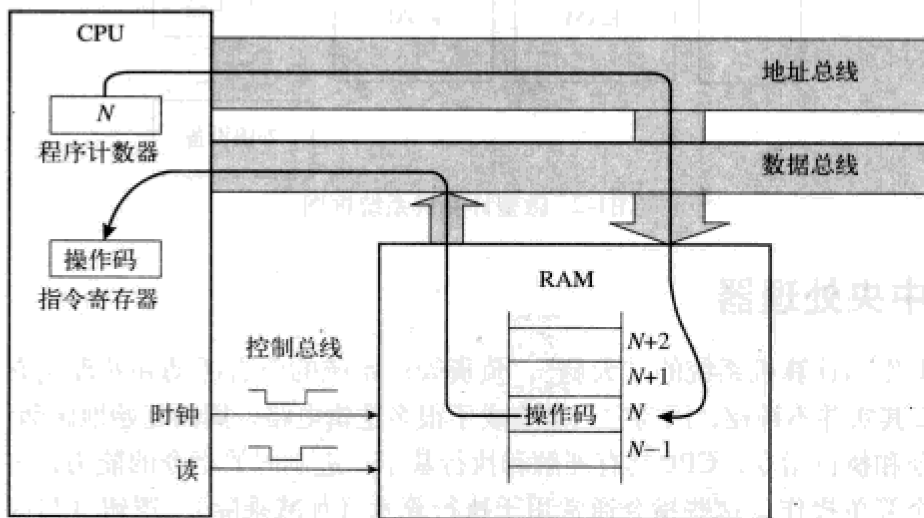


图1-4 一个指令读取周期内总线的相关动作

在执行阶段，CPU对操作码进行解码（或判读）并产生控制信号，在内部寄存器和算术逻辑单元之间交换数据，通知算术逻辑单元执行指定的操作。由于可执行的操作多种多样，所以这样的解释有局限性。这样的解释对于类似于“寄存器加1”这样简单操作是适用的，而复杂的指令则需要更多的步骤才能完成，例如为了运算需要读取第2字节和第3字节作为待运算的数据。

组合在一起能够完成一种有意义任务的指令序列称为**程序**或**软件**，程序才是真正的核心所在。任务完成的效率与准确程度主要取决于软件的编写质量而非CPU的复杂程度。程序控制着CPU的运行，但也会偶尔出现故障，正如它们的设计者也有弱点一样。类似“计算机出错了”这样的话是一种误导，因为尽管设备的故障不可避免，但错误常常是由程序上的不足或操作失误造成的。

1.4 半导体存储器：RAM和ROM

在计算机系统中，程序和数据存储在存储器中。计算机存储器的种类繁多，相

关的术语也非常丰富。由于技术更新非常频繁，只有做广泛持续的研究才能跟上最新发展的步伐。可供CPU直接访问的存储器有两类半导体集成芯片，分别为RAM和ROM。二者有两个明显区别：第一，ROM是只读存储器，而RAM是可读写存储器；第二，RAM有易失性（即掉电后存储的内容消失），而ROM具有非易失性。

大多数计算机系统具有磁盘驱动器和较小容量的ROM，恰好满足用于存储短小的、频繁使用的执行输入输出操作的软件程序。用户程序和数据存放在磁盘中，在执行时再读到RAM中。随着RAM每字节成本的持续下降，小型计算机系统常常拥有几兆字节的RAM。

1.5 总线：地址总线、数据总线和控制总线

总线是一组传送有共同目的的信息的线路的集合。访问CPU外围设备需要3种总线：**地址总线**、**数据总线**和**控制总线**。对每个读或写操作，CPU都会将数据（或指令）在存储器中的地址放到地址总线上，然后在控制总线上发送一个读或写的信号。读操作将从存储器的指定位置读出数据并放到数据总线上，然后CPU读取该数据并存放到一个内部寄存器中；写操作是CPU将数据输出到数据总线上，存储器根据CPU的控制信号识别出写周期操作并将数据存储到指定位置。

大部分小型计算机有16条或20条数据总线。如果有n条数据总线，每条总线上的电平可能是高（1）或低（0），那么可以访问2^n个不同的地址单元。16位的地址总线可以访问$2^{16}=65\ 536$个地址，20位的地址总线可以访问$2^{20}=1\ 048\ 576$个地址。缩写K（10^3）表示$2^{10}=1024$，所以16位总线可以寻址$2^6\times2^{10}=64$K个地址，20位总线可以寻址$1024\times1024=1024$K或者1M空间。其中的缩写M（1兆）表示$2^{20}=1024\times1024=1024\text{K}=1\ 048\ 576$。

数据总线在CPU和存储器或CPU和输入/输出设备之间传输数据。对计算机各种操作所消耗的宝贵的执行时间的比重做了大量的研究工作，结果表明，简单的传送数据操作占用了2/3的执行时间。由于主要的数据传送操作发生在CPU寄存器和外部RAM/ROM之间，所以数据总线的数目（总线宽度）对于计算机的整体性能是非常重要的。总线宽度的限制是一个瓶颈：假设一个计算机系统拥有大量的存储器，CPU的计算能力非常强大，但是通过数据总线在CPU和存储器之间传送数据时，由于总线的宽度而受限，此时数据总线的宽度就成为访问数据性能的瓶颈。

由于数据总线的重要性，其位数已经成为标志计算机性能的重要指标。“16位计算机”即表明该计算机具有16条数据总线。大部分计算机按照4位、8位、16位或32位分级，计算性能随着数据总线位数的增加而增强。

数据总线是双向的（见图1-2），而地址总线是单向的。地址信息总是由CPU提供（图1-2中箭头所指的方向），然而数据可以双向传输，其方向取决于是读操作还

是写操作①。注意，这里所说的“数据”是广义的，指在数据总线上传送的所有“信息”，既可能是程序的指令，又可能是指令中附加的地址，还可能是程序要使用的数据。

6

控制总线是各种控制信号的组合，每个信号都有自己的功能，控制系统活动的有序进行。通常控制信号为CPU所提供的时序信号，用于同步数据和地址总线的信息传送活动。虽然通常针对CPU和存储器之间的数据传送操作只有3种信号，如时钟（CLOCK）、读（READ）和写（WRITE），但不同CPU的控制信号的名称和功能也不相同，具体细节需要参考制造商提供的相关说明书。

1.6 输入/输出设备

输入/输出设备（即“计算机外围设备”）提供了计算机系统和“现实世界”之间的通信通道。如果没有输入/输出设备，计算机系统只是一个封闭的机器，对使用者来说没有什么用处。下面介绍3种输入/输出设备：**大容量存储设备、人机交互设备和控制/监测设备**。

1.6.1 大容量存储设备

如同半导体RAM和ROM一样，大容量存储设备在存储技术的舞台上扮演着重要角色，而且其技术也在不断发展和进步。大容量存储设备可以保存大容量的信息（程序或数据），而如此大容量的信息无法存放在计算机的小容量的RAM或“主”存储器里。CPU在访问这些数据之前必须先将其读取到主存储器中。按访问的方便程度，大容量存储设备分成**联机型**和**档案型**两类。联机型存储器（比如磁盘）是指CPU可以根据程序的需要在无人工干预的情况下访问的存储器类型；档案型存储器用于保存不常使用而且需要手工参与才能将其内容读取到系统中的数据。尽管光盘（如CD-ROM或WORM②技术）问世并且以其稳定可靠、大容量和低成本的优点而有可能改变档案型存储器的传统概念，但档案型存储器通常仍是指磁带或磁盘。

1.6.2 人机交互设备

人和机器的沟通是通过人机交互设备实现的，最常见的有视频显示终端和打印机。打印机是纯粹的输出设备（产生硬复制输出），而视频显示终端实际包含两个设备，包括用于输入的键盘和用于输出的CRT（阴极射线管）显示器。被称为“人体工学”或“人性因素”的整个工程领域已逐渐发展成这种模式，即在设计计算机外围设备时必须考虑人的因素，目的是保证人们能够安全、舒适、高效地使用计算

① 地址信息有时也通过直接存储器访问（DMA）方式提供（CPU除外）。

② CD-ROM即小型只读存储器，WORM即单次写多次读存储器。1张CD-ROM包含700MB的存储空间，能装下整个32卷《不列颠百科全书》。

机。事实上，生产这类外围设备的公司要比生产计算机的多很多。大部分计算机系统至少都有3个外围设备：键盘、CRT显示器和打印机。其他人机交互设备有游戏杆、光笔、鼠标、麦克风和扬声器。

1.6.3 控制/监测设备

通过控制/监测设备（以及精心设计的接口电路和软件），计算机可以执行各种面向控制的任务。计算机可以不间断工作而且不会疲劳，远远超过人类的能力极限。许多应用场合，如楼宇温度控制、家庭保安、电梯控制、家电控制甚至汽车的零件焊接等，都有可能用到这类设备。

控制设备是输出设备（即**执行机构**）；当以电压或电流驱动时（如电机和继电器），控制设备可以对周围的环境施加影响。监测设备是输入设备（即**传感器**），它受外界的热、光、压力、运动等的激励，并把这种能量转换为计算机可读的电压或电流信号（如光电二极管、热敏电阻和开关）。接口电路负责把电压或电流信号转换为二进制数据，或反过来将二进制数据转换为电压或电流信号。软件负责分析输入信号并产生相应的输出信号。这些设备与微控制器之间的硬件和软件接口是本书的主要论题之一。

1.7 程序：大程序和小程序

前面的讨论主要集中在计算机系统的硬件方面，只是偶尔提到能使其工作的程序或软件。近年来，硬件与软件的相对重要性发生了戏剧性的转变。在计算机发展的早期，人们对计算机硬件的物料、生产制造和维护成本的关注远远超过对软件成本的关注。现在，随着大规模集成电路芯片的大批量生产，硬件成本已经不是最主要的了，软件成本在计算机自动化应用中占到了很大的比重，因为编程、编写文档以及软件维护、升级和发布都属于劳动密集型工作。

下面讨论软件的不同类型。图1-5描述了在用户与计算机系统硬件之间的3个软件层次：**应用软件**、**操作系统**和**输入/输出子例程**。

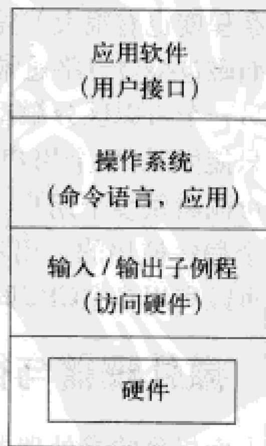


图1-5 软件的层次

输入/输出子例程位于最底层，直接对硬件操作，完成诸如从键盘读取字符、在CRT上显示字符、从磁盘上读取信息等任务。由于这些子例程与硬件的关系非常紧密，所以通常由硬件设计人员编写，而且通常存储在ROM中［例如IBM PC中的BIOS（基本输入/输出系统）］。

为了给程序员提供访问系统硬件的方法，输入/输出子例程定义了非常明确的进入和退出条件。只需要

初始化CPU寄存器并调用输入/输出子例程，子例程就自动执行并将运行结果返回到CPU的寄存器或保留在系统RAM中。

除了是作为输入/输出子例程的完整补充，ROM还有一个启动程序，在系统上电或用户手动复位的时候运行。ROM的非易失性在这里显得非常重要，因为必须保证程序在系统上电时存在。初始化内存、系统自我诊断等任务都是由启动程序完成的，为系统启动做好准备。最后，**引导装入程序**将磁盘的第1磁道上的内容（1个小程序）读入RAM并将控制权交给它。程序随即从磁盘载入操作系统（1个大程序）常驻RAM中的部分，再把控制权交给操作系统，系统启动完成。这个过程可以总结为“系统通过自有的引导程序实现了自身的启动”。

8

操作系统是依附于计算机系统的一个大程序集，它提供了访问、管理并有效利用计算机资源的机制。这些功能通过操作系统的**命令语言**和**实用程序**实现，这种方法促进了应用软件的发展。只要应用软件设计得好，使用者只需很少了解或者无需了解操作系统的细节就可以方便地使用计算机。提供高效、有价值、安全的用户界面是应用软件设计的基本目标。

1.8 微型机、小型机和大型机

按照体积和计算能力，计算机可分为微型机、小型机和大型机。微型机的主要特征在于CPU的大小和封装：它的CPU是1个单芯片——**微处理器**。小型机和大型机的CPU都由多个芯片组成，其他部分的结构也要复杂一些。小型机通常拥有数个芯片，而大型机包括数块电路板上的大量芯片，这是要获得更高的工作速度和更强的计算能力所必需的。

典型的微型机（如IBM的PC机、苹果的Macintosh和Commodore的Amiga）都采用了1个微处理器作为其CPU。RAM、ROM和接口电路也需要使用许多集成电路。通常，所需元件的数目随着计算能力的增强而增加。输入/输出设备不同，接口电路的复杂程度也有很大差异。例如，许多微型机中的扬声器只需要1对逻辑门就可以驱动，而磁盘接口电路通常包含许多集成电路芯片，有些还是大规模集成电路芯片。

9

另一个可以区分微型机、小型机和大型机的特征是：微型机是单用户单任务系统，只能同时供1个用户使用，运行1个程序。而小型机和大型机是多用户多任务系统，可同时供多个用户使用，运行多个程序。实际上，CPU资源是按时间分段分配给各个程序使用的，用户感觉程序在同时运行只是一种错觉。（但在多处理器系统中，可以利用多个CPU同时处理各自的任务。）

1.9 微处理器与微控制器的比较

上面提到的微处理器是微型机中使用的单芯片CPU。那么，微控制器和微处理

器之间有区别吗？下面从**硬件体系结构**、**应用领域**和**指令集特征**3个方面讨论这个问题。

1.9.1 硬件体系结构

为了突出微控制器和微处理器之间的区别，图1-6中补充了图1-2中没有画出的细节。

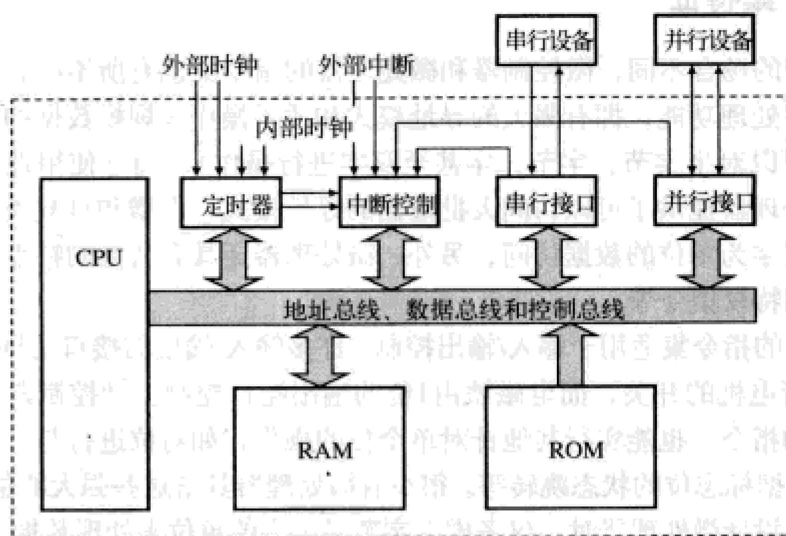


图1-6 微型计算机的系统结构框图

微处理器是一个单芯片CPU，而微控制器则在一块集成电路芯片中集成了CPU和其他电路，构成了一个完整的微型计算机系统。图1-6虚线框中所示的是大多数微控制器的完整结构。除了CPU，微控制器还包括RAM、ROM、串行接口、并行接口、定时器和中断调度电路，这些都集成在同一块集成电路上。虽然片上RAM的容量比普通微型计算机系统还要小，但是这并未限制微控制器的使用。在后面可以了解到，微控制器有着非常广泛的应用范围。

微控制器的一个重要特征是内建的中断系统。作为面向控制的设备，微控制器经常要实时响应外界的激励（中断）。微控制器必须执行快速上下文切换，挂起一个进程去执行另一个进程以响应一个事件。例如，打开微波炉的门就是一个事件，在基于微控制器的产品中这个事件将触发一个中断。微处理器也能拥有强大的中断功能，但是通常需要外部元件配合，而微控制器在片上集成了所有处理中断必需的电路。

1.9.2 应用领域

微处理器在微型机系统中通常作为CPU使用。其设计正是针对这样的应用，这也是微处理器的优势所在。然而，微控制器通常用于面向控制的应用。其系统设计

追求小型化，尽可能减少元件数量。在过去，这些应用通常需要用数十个甚至数百个数字集成电路来实现。使用微控制器可以减少元器件的使用数量，只需要一个微控制器、少量外部元件和存储在ROM中的控制程序就能实现同样的功能。微控制器适用于那些以极少元件实现对输入/输出设备进行控制的场合，而微处理器适合在计算机系统中进行信息处理。

10

1.9.3 指令集特征

由于应用的场合不同，微控制器和微处理器的指令集也有所不同。微处理器的指令集增强了处理功能，拥有强大的寻址模式和适于操作大规模数据的指令。微处理器的指令可以对半字节、字节、字甚至双字进行操作①。通过使用地址指针和地址偏移，微处理器提供了可以访问大批数据的寻址模式。自增和自减模式简化了以字节、字和双字为单位的数据访问。另外，微处理器还具有其他的特点，如用户程序中无法使用特权指令等。

11

微控制器的指令集适用于输入/输出控制。许多输入/输出的接口是1位的。例如，电磁铁控制着电机的开关，而电磁铁由1位的输出端口控制。微控制器具有设置和清除单个位的指令，也能实行其他针对单个位的操作，如对位进行与、或、异或的逻辑运算，根据标志位的状态跳转等。很少有微处理器具备这些强大的位操作能力，因为设计者在设计微处理器时，仅考虑以字节或更大的单位来处理数据。

在对设备的控制和监测方面（可能是通过1位的接口），微控制器具有专门的内部电路和指令用于输入/输出操作、事件定时、启用以及设置由外部激励导致的中断的优先权。微处理器一般需要配合附加的电路（串行接口芯片、中断控制器、定时器等）才能执行类似的任务。不过，单纯就处理能力而言，微控制器永远达不到微处理器的水平（在其他条件相同的情况下），因为微控制器芯片中的集成电路的很大一部分用于实现其他的片上功能，当然，代价就是牺牲一部分处理能力。

由于微控制器中片上资源非常紧张，它的指令必须相当精简，大部分指令的长度都不超过1个字节。控制程序的设计原则通常是要求程序能够在片上ROM装得下，因为即使只扩展1片外部ROM也将显著提高最终产品的硬件成本。微控制器指令集的基本特点就是具有精简的编码方案。微处理器不具备这样的特点，因为它们强大的寻址方式使得指令编码不够简洁。

1.10 新概念

微控制器与其他取得惊人发展的产品一样，是在市场需求与新技术这两股互相推动的力量共同促进下出现的。新技术方面，前面已经提到，半导体工艺的发展使

① 更为通用的解释是：4位=1半位元组，8位=1字节，16位=1字，32位=1双字。

得在更小的面积上集成更多的晶体管和低成本的大批量生产成为可能。市场需求方面，工业界和消费者则需要更加精密复杂的工具和娱乐产品[①]，这种需求在许多领域都存在。最典型的例子也许就是汽车上的仪表盘。过去的十年中，汽车的控制中心（仪表盘）的演变反映了微控制器及其他技术的发展。司机们曾经满足于了解自己的行驶速度，而现在他们可以看到仪表盘上显示的燃油消耗情况和预计到达时间。以前觉得能够知道发动车辆的时候哪个安全带没扣上就足够了，现在汽车则会告知司机哪个安全带出现了故障。如果车门没有锁紧（也许是门夹住了安全带），就会得到适时的语音提示。

12

这里有必要说明一下，微处理器（这里也可包括微控制器）被戏称为"问题的解决方案"。它们在降低消费类产品电路的复杂程度方面的作用很显著，制造商们通常不希望在产品中添加一些多余的功能，因为这很容易实现。如果这样做，通常会适得其反，往往是开始大受欢迎，但最终却招人厌烦。典型的例子是，最近的语音产品都出现了附加额外特殊性能的设计趋势。无论是汽车、玩具还是烤箱，它们都常被当成低水平过度设计的例子，类似于20世纪80年代的艺术装饰品。但是可以确信，随着时间的推移，这些表面的东西会逐渐失去生命力，只有精致实用的产品才会留存下来。

微控制器非常特殊，它们不是用于计算机中，而是用于工业产品和消费类产品中。使用这些产品的人们通常察觉不到微控制器的存在，对于他们来说，产品内部的元件只是无关紧要的设计细节。微波炉、程控恒温器、电子秤还有汽车都是这样的例子。在这些产品内部，电子元件将微控制器与面板上的按钮、开关、灯和报警器连接在一起；对于用户来说，除了少数新增加的功能外，他们操作的方法和使用旧的机电结构的产品是一样的，用户感觉不到微控制器的存在。

计算机系统拥有编程和反编程的能力，与之不同，微控制器的程序只能固定地执行某一项任务。这也使得两者的体系结构有很大差异。计算机系统的RAM要比ROM大得多，用户程序在相对较大的RAM中运行而硬件接口例程在ROM中运行。相反，微控制器的ROM要比RAM大得多。控制程序相对较大，存储在ROM中，而RAM只是用于临时存储。由于控制程序永久性地存储在ROM中，因此也被称为固件。从持久性来说，固件介于软件（RAM中的程序，断电后会消失）和硬件（物理电路）之间。软件和硬件之间的这种差别有些类似于纸张（硬件）和写在纸上的字（软件）。固件则可以比喻为一封为了特定目的而设计印刷的标准格式的信件。

1.11 得与失：设计范例

微控制器所完成的任务并不是全新的，新颖的是现在的设计需要的元件数目比

① "'市场需求'实际是'市场导向'"这一观点有时是有争议的，"市场导向"将激励技术自我发展和进步。

以前减少了。以前需要数十甚至数百个集成电路才能完成的设计，现在只要少量元件和一个微控制器就行了。微控制器的可编程能力和高集成度是元件数量减少的直接原因，这通常可以缩短开发周期、降低制造成本，还可以降低产品的功耗，提高产品的可靠性。以前需要多个集成电路才能完成的逻辑操作，现在通常可以在微控制器内由控制程序完成。

13

获得这些好处是以牺牲处理速度为代价的。基于微控制器的方案永远没有基于分立元件的方案那么快。需要对事件有极高响应速度的解决方案（少数应用场合）很难用微控制器实现。让我们来考查一下，图1-7所示的用8051微控制器实现的与非门。

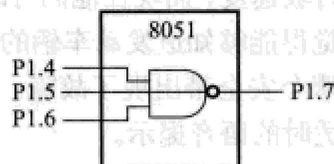


图1-7 用微控制器实现的简单逻辑电路

读者似乎不容易想到可以用微控制器来实现这样的功能，不过确实可以。程序的流程图如图1-8所示。用8051汇编语言编写的程序如下。

```
LOOP:   MOV    C,P1.4      ;READ P1.4 BIT INTO CARRY FLAG
        ANL    C,P1.5      ;AND WITH P1.5
        ANL    C,P1.6      ;AND WITH P1.6
        CPL    C           ;CONVERT TO "NAND" RESULT
        MOV    P1.7,C      ;SEND TO P1.7 OUTPUT BIT
        SJMP   LOOP        ;REPEAT
```

如果在8051微控制器上运行这个程序，可以实现3输入的与非门功能（可用电压表或示波器验证）。与TTL电路相比，从输入到正确输出的传输延时非常长。从接收到输入端的电压变化开始计算，延时大概是3ms~17ms（假设是标准的8051，晶振频率12MHz）。完成相同的功能的TTL电路，延时是10ns量级，比8051快3个数量级。显然，为实现同一功能微控制器在速度方面无法与TTL电路竞争。

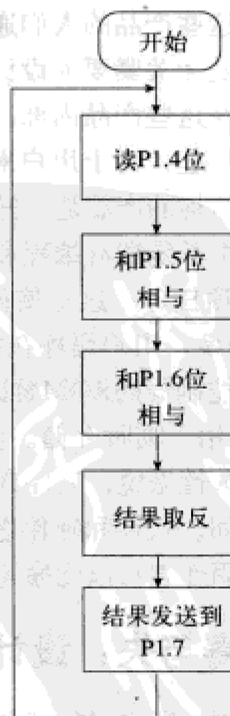


图1-8 逻辑门程序的流程图

但是在许多应用场合，特别是与人工操作相关的应用，延时是纳秒还是微秒或毫秒量级并不重要。（难道司机需要汽车每毫秒报告一次油压下降了多少吗？）逻辑门的例子说明了微控制器具有执行逻辑操作的能力。当设计变得复杂的时候，基于微控制器的设计就显现出优势来。正如前面提到的，减少元件数量会带来很多好处，而且，由控制程序实现功能，只需要改动软件就可完成功能上的调整。在产品制造周期中，这种调

整方式可以将相关影响降到最低。

关于控制器的介绍就到这里。下一章将详细介绍MCS-51系列的微控制器。

14

习题

1.1　第一种被广泛使用的微处理器是什么？是在哪一年由哪家公司发布的？

1.2　20世纪70年代，两家小微处理器公司——MOS技术公司和Zilog公司也推出了各自的微处理器芯片，写出这两款微处理器芯片的名字。

1.3　8051微控制器是在哪一年发布的？其上一代产品是什么？哪一年发布的？

1.4　给出本章中讨论的两种类型的半导体存储器的名称。哪种类型的存储器可以在断电后保持其中的信息不丢失？通常用什么术语来描述这种性质？

1.5　CPU中哪个寄存器总是用于存储地址？这个寄存器中存储的是什么地址？

1.6　在获取操作码阶段，地址总线和数据总线上的信息是什么？这时这些总线上的信息是朝着哪个方向传输的？

1.7　具有18位地址总线和8位数据总线的计算机的寻址范围有多大？

1.8　“16位计算机”中的“16位”是什么意思？

1.9　联机存储和档案存储的区别是什么？

1.10　除了磁带和磁盘之外，还有哪种技术可用于档案存储？

1.11　对于计算机系统，人体工学的目标是什么？

1.12　有如下几种人机交互设备：游戏手柄、光笔、鼠标、麦克风、扬声器。哪些是输入设备，哪些是输出设备？

1.13　本章提到的软件的三个层次中，哪个是最低的层次，这个层次的软件的功能是什么？

1.14　执行机构和传感器的区别是什么？请各举一例说明？

1.15　什么是固件？比较基于微控制器的系统和基于微处理器的系统，哪一种更依赖于固件？为什么？

1.16　微控制器的指令集和微处理器的指令集有所不同的一个重要特征是什么？

1.17　试举出5种本章中没有提到的、在未来可能使用微控制器的产品。

15
~
16

第2章 硬 件 概 述

2.1 MCS-51™系列简介

MCS-51™是由Intel公司开发、生产并推向市场的一个微控制器集成芯片系列。其他芯片制造商，如西门子、AMD、富士通、飞利浦等，也获得了生产MCS-51™系列微控制器的许可证，成为该产品的第二供应商。MCS-51™系列的每种型号微控制器在大体相似的基础上又都各具特点，以满足不同的设计需求。

本章介绍MCS-51™系列微控制器的硬件体系结构。附录E中给出了Intel公司入门级设备（如8051AH）的数据表。该附录还介绍了这些设备的特征细节（如电气特性）。

本章在描述硬件特性时，在许多地方是结合简短的指令程序来进行的，并给出了每个例题的简单解释。关于指令集的完整细节内容将在第3章中介绍，也可以参考附录A的8051指令速查，或者通过查询附录C来了解每条8051指令的定义。

8051是通用的MCS-51™系列芯片，也是51系列最先推向商用的型号。8051微控制器的主要特征如下：

- 4KB ROM（工厂掩模预编程）
- 128B RAM
- 4个8位I/O（输入/输出）端口
- 2个16位定时器
- 1个串行接口
- 64KB外部程序存储器空间
- 64KB外部数据存储器空间
- 布尔逻辑处理器（可进行单个位处理）
- 210位可寻址空间
- 4μs乘/除法

MCS-51™系列的其他型号提供了不同大小的片上ROM/EPROM、片上RAM或者3个定时器的组合。同时，每种型号还提供低功耗的CMOS版本（如表2-1所示）。

本书使用8051泛指MCS-51™系列的微控制器。如果讨论某种型号基于8051的增强功能，将会明确指出其具体型号。图2-1是8051的结构框图，上面提到的各种特征都包含在内（见附录D）。

表2-1 MCS-51™系列芯片的比较

型号	片上程序存储器	片上数据存储器	定时器
8051	4KB ROM	128B	2
8031	0	128B	2
8751	4KB EPROM	128B	2
8052	8KB ROM	256B	3
8032	0	256B	3
8752	8KB EPROM	256B	3

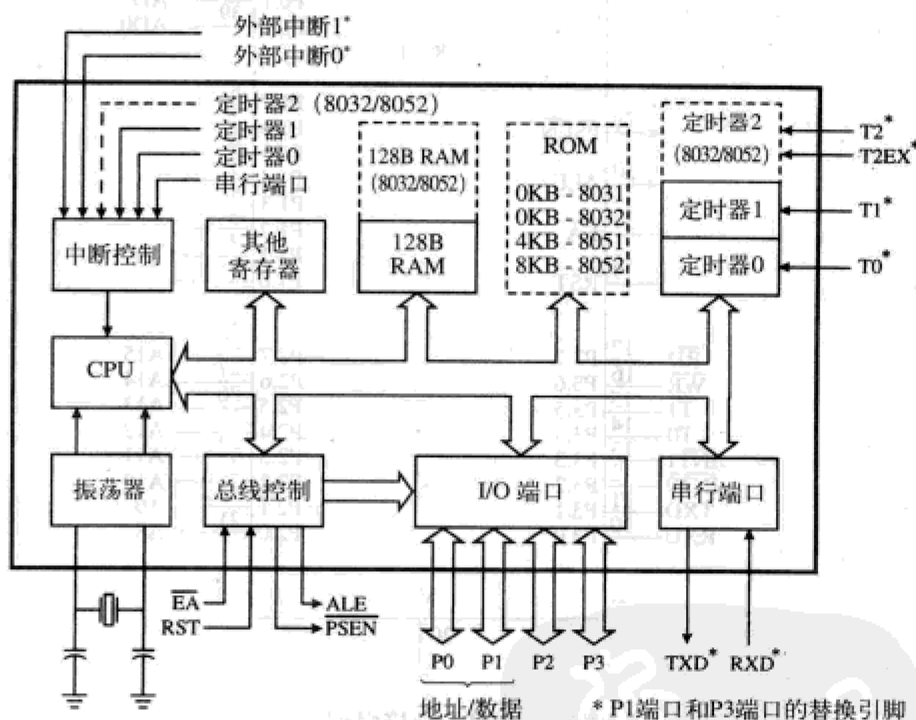


图2-1 8051结构框图

2.2 引脚

本节通过8051的一个外部特性（外接引脚）介绍其硬件结构（见图2-2）及其功能，下面简要说明各引脚的功能。

如图2-2所示，8051的40个引脚中有32个作为I/O端口引线。在这32条引线中，有24条具备两种用途（在8032/8052中是26条），它们既可以作为标准的I/O端口，也可以作为控制信号线或地址总线及数据总线的一部分。

在进行需要最少外部存储器或其他外部元件的设计中，这些端口被作为通用I/O端口使用。每个端口的8条引线可作为一个整体使用，用于与打印机和数模转换器等并行设备连接；各条端口线还可以单独使用，与开关、LED、晶体管、电磁铁、

电动机、扬声器等只需单个位就可控制的设备连接。

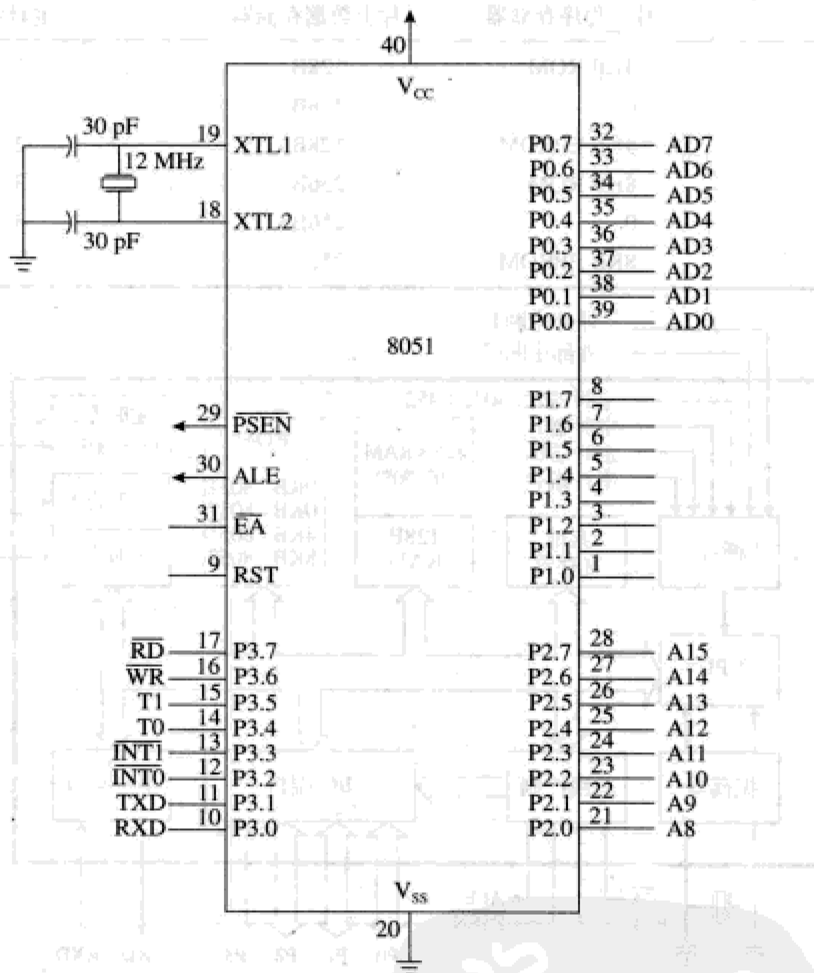


图2-2　8051的外接引脚

2.2.1　端口0

端口0占据8051的第32号~第39号引脚，是一个双用途端口。在元件最少化的设计中，该端口作为通用I/O端口使用。对于有外部存储器的较大设计，端口0充当多路复用的地址总线和数据总线（见2.7节）。

18

2.2.2　端口1

端口1是专用I/O端口，位于1~8号引脚，端口信号线命名为P1.0、P1.1、P1.2等。这些引脚用作接口与需要的外部设备连接，没有被分配第二功能，因此，它们仅作为与外部设备的接口。在8032/8052芯片中情况有所不同，P1.0和P1.1除了用作通用I/O引线外，还可以作为第3个定时器的外部输入。

2.2.3 端口2

端口2（引脚21~28）是个双用途端口，一方面可以作为通用I/O端口，另一方面作为地址总线的高字节，用于外部程序存储器或多于256B的外部数据存储器（请参考2.7节）。

2.2.4 端口3

端口3占据10~17号引脚，是个双用途端口。除了可作为通用的I/O端口外，端口3的引脚都是多功能引脚，每个引脚还具备其他与8051具体特性相关的功能。端口3和端口1中各个引脚的附加功能如表2-2所示。

表2-2 端口引脚的附加功能

位	信号名	位地址	第二功能
P3.0	RXD	B0H	接收串行端口数据
P3.1	TXD	B1H	发送串行端口数据
P3.2	$\overline{\text{INT0}}$	B2H	外部中断0
P3.3	$\overline{\text{INT1}}$	B3H	外部中断1
P3.4	T0	B4H	定时器/计数器0外部输入
P3.5	T1	B5H	定时器/计数器1外部输入
P3.6	$\overline{\text{WR}}$	B6H	外部数据存储器写选通
P3.7	$\overline{\text{RD}}$	B7H	外部数据存储器读选通
P1.0	T2	90H	定时器/计数器2外部输入
P1.1	T2EX	91H	定时器/计数器2捕获/重载

2.2.5 $\overline{\text{PSEN}}$

8051有4个专用总线控制信号。$\overline{\text{PSEN}}$（程序存储启用）信号是位于引脚29的输出信号，用来控制启用外部程序（代码）存储器。它通常连接到EPROM（可擦可编程只读存储器）的$\overline{\text{OE}}$（输出使能）引脚，EPROM收到信号后就允许8051读取其中的程序。

在从存储在外部程序存储器读取指令期间，$\overline{\text{PSEN}}$信号保持低电平。EPROM中存储的程序的二进制代码（操作码）被读出，通过数据总线被锁存入8051的指令寄存器中等待译码。在执行内部ROM（8051/8052）中的程序时，该信号无效（高电平）。

2.2.6 ALE

ALE（地址锁存启用）信号通过引脚30输出，对于使用过Intel 8085、8088和8086微处理器的人来说，应该非常熟悉该信号的功能。8051同样也使用ALE信号来

多路复用地址总线和数据总线。当端口0工作于第二功能模式——作为数据总线和低8位地址总线使用时，在存取周期的前半个周期，总线上传输的是地址的低字节，ALE信号将总线上的地址信息锁存到外部寄存器，在存取周期的后半个周期，当发生数据传输时，端口0用于输入/输出数据（见2.7节）。

ALE信号的频率是片上振荡器工作频率的1/6，对于系统的其余部分，可作为通用时钟信号使用。如果8051由12MHz的晶振驱动，那么ALE信号的振荡频率是2MHz。唯一的例外发生在执行MOVX指令时，会丢失1个ALE脉冲（见图2-11）。在有EPROM的8051芯片上，该引脚还用来作为编程输入信号。

2.2.7 $\overline{EA}$

$\overline{EA}$（外部访问）信号通过引脚31输入，通常被接到高电平（+5V）或低电平（地）上。如果$\overline{EA}$为高电平，则8051/8052在访问存储器的低4KB/8KB的同时，从内部ROM执行程序。如果为低电平，CPU仅执行位于外部存储器中的程序（$\overline{PSEN}$信号应为低电平）。在8031/8032芯片上，$\overline{EA}$必须为低电平，因为它们没有片上程序存储器。如果8051/8052上的$\overline{EA}$为低电平，则内部ROM被禁用，CPU仅执行外部EPROM上的程序。对于具有EPROM的8051芯片，把$\overline{EA}$引脚接到+21V电压（V_{pp}）上时可进行内部EPROM编程。

21

2.2.8 RST

RST（复位）信号通过引脚9输入，是8051的主复位信号。如果此引脚维持至少2个机器周期的高电平，那么，8051内部寄存器将会被配置为默认的初始值，使得系统重新启动。正常工作时，RST必须保持低电平（见2.9节）。

2.2.9 片上振荡器输入

如图2-2所示，8051包含一个片上振荡器，要驱动该振荡器，典型方式是在引脚18和19之间连接1片晶振，而且驱动电路中还需要2个稳定电容。

虽然如80C31BH-1的晶振频率可高达16MHz，但MCS-51™系列的大多数芯片常用的晶振频率是12MHz。片上振荡器不一定非得要晶振驱动。如图2-3所示，TTL时钟信号还可以通过XTAL1和XTAL2引脚与8051连接。

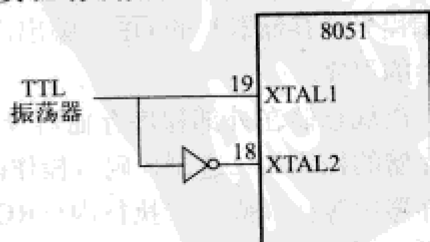


图2-3 利用TTL振荡器驱动8051

2.2.10 电源

8051需要一个+5V电源供电，V_{CC}接在第40引脚上，V_{SS}（地）接在第20引脚上。

2.3 I/O端口结构

图2-4简要描述了端口引脚的内部电路。向端口的一个引脚写入数据时，数据首先被载入端口引脚的锁存器中，锁存器驱动一个连接到该端口引脚的场效应管。端口1、2、3可以驱动4个低功率的肖特基TTL负载，端口0可以驱动8个LS TTL负载（详情请参考附录E）。需要注意的是，端口0的引脚没有负载电阻（但作为外部地址/数据总线时除外），因此，根据该引脚所驱动的设备的输入特性，决定是否需要外部负载电阻。

8051具有“读锁存器”和“读引脚”两种能力。在引脚负载很大的情况（如驱动晶体管的基极）下，在执行“读取－修改－写入”一类的指令（如CPL P1.5）时需要从锁存器中读取数据，以免错误地判断引脚电平。从端口的单个位输入数据的指令（如MOV C, P1.5）则需要读取引脚上的数据。这时，端口锁存器必须被置1，否则，输出场效应管导通，会拉低引脚上的高输出电平。系统复位时，会把所有端口锁存器置1，然后可以直接使用端口引脚作为输入而无需再明确设置端口锁存器。但是，如果端口锁存器被清除（如CLR P1.5），就不能把该端口直接作为输入使用，除非先把对应的锁存器置为1（如SETB P1.5）。

图2-4中，没有描述端口0、端口2和端口3附加功能所使用的电路。当使用附加功能时，端口2的输出被切换为内部地址信号，端口0的输出被切换为地址/数据信号，端口3的输出被切换为控制信号。

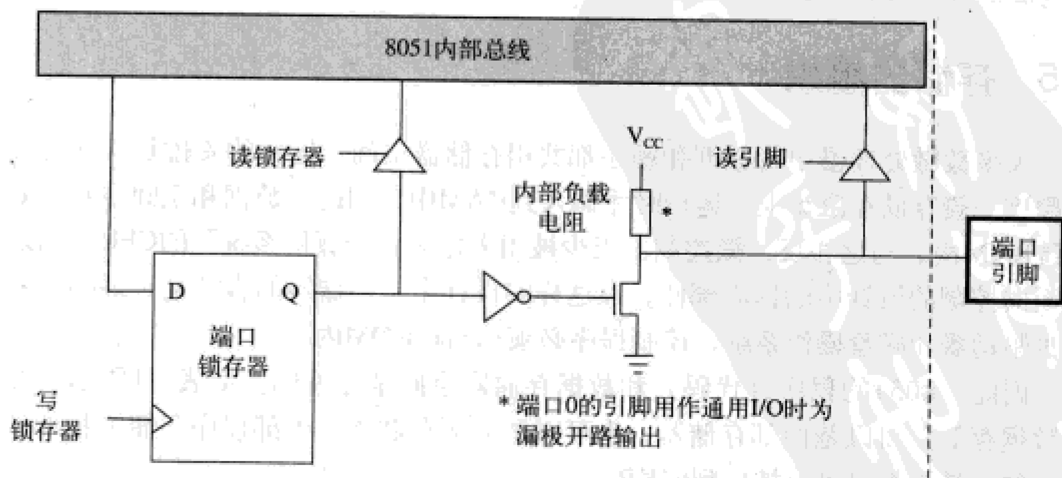


图2-4 I/O端口电路结构

2.4 时序和机器周期

8051的片上振荡器由接在18脚和19脚的外部石英晶体振荡器驱动。晶体的典

型频率为12MHz，也就是说每秒钟产生12 000 000个时钟周期。这些振荡时钟周期构成了8051芯片的时序和同步基础：所有由8051执行的操作都是以这些周期进行的。

以这个振荡时钟为参照，8051需要两个时钟周期来执行一个单独的操作，包括指令的读取、解码和执行。两个时钟周期的持续时间被称为一个**状态**。所以，为了完整地执行一条指令，8051一般需要6个状态或12个时钟周期的时间，因为在执行指令前需要先进行获取和解码操作。6个状态的时间也称为1个**机器周期**。当然，更复杂的指令需要花费1个以上的机器周期才能执行完毕。附录B和C列出了8051指令集的每条指令执行所需要的机器周期数，机器周期的个数介于1和4之间。图2-5表示出了振荡时钟周期（P）、状态（S）及机器周期的相互关系。



图2-5 振荡时钟周期、状态和机器周期的关系

一般情况下，8051的片上振荡器受12MHz的外部晶振驱动，1个时钟周期为$T_{clock} = 1/f_{osc} = 1/12\ \text{MHz} = 83.33\ \text{ns}$. 1个机器周期包括12个时钟周期，所以其时间长度为$83.33\ \text{ns} \times 12 = 1\ \mu\text{s}$。

23

2.5 存储器组织

大多数微处理器中的数据和程序都共用存储器空间。这样的安排是合理的，因为程序一般存储在磁盘上，运行时才调入到RAM中，因此，数据和空间同时存储在系统RAM内。与之相反，微控制器很少被用来作为“计算机系统”的CPU，而是作为面向控制的设计中的核心部件。在这样的设计中，存储器的容量是有限的，没有磁盘驱动器和磁盘操作系统，控制程序必须存储在ROM内。

因此，8051的程序（代码）和数据存储器空间是分开的。如表2-1所示，程序和数据存储器可以是内部存储器，也可以是外部存储器，外部程序存储器和外部数据存储器最大都只可以扩展到64KB。

内部存储器由片上ROM（仅8051/8052）和片上数据RAM组成。片上RAM包括通用存储器、可位寻址存储器、寄存器组以及特殊功能寄存器等。

微控制器的存储器有两个值得注意的特点：

(1) 各寄存器和输入/输出端口在存储器空间中以内存映射的形式出现，可像访问其他存储器空间一样访问这些寄存器和输入/输出端口；

(2) 栈存储在内部RAM中，而在微处理器中通常把栈存储在外部RAM中。

图2-6概要描述了不带ROM的8031系统的存储空间分布，这里略去了片上数据存储器的细节（8032/8052的增强部分将在后面介绍）。

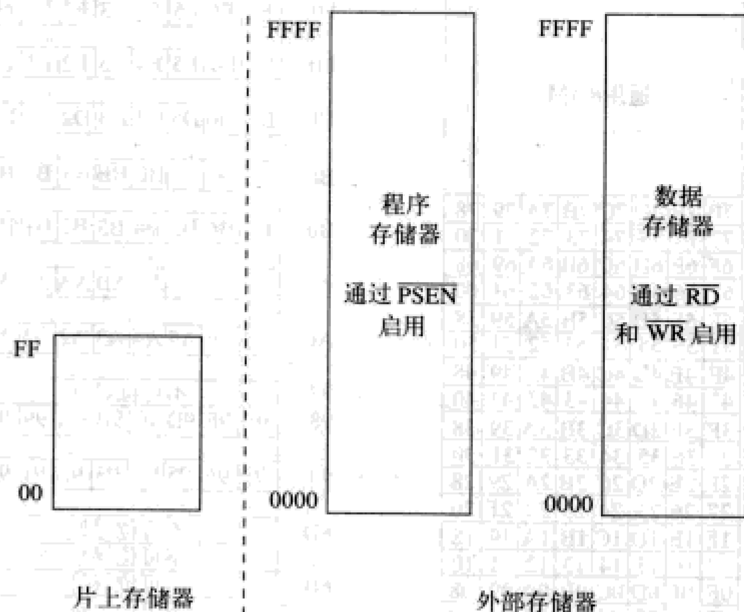


图2-6 8031存储器空间结构图

图2-7给出了片上数据存储器的细节。内部数据存储器空间分为**内部RAM**（00H~7FH）和**特殊功能寄存器**（80H~FFH）。内部（片上）数据存储器和内部RAM是一对容易混淆的概念。8051的内部数据存储空间的地址范围是00H~FFH，计256B。但是只有低半部分（00H~7FH）可用于存放通用数据，而高半部分（7FH~FFH）大多有特殊用途而不能存放通用数据。因此，只有低半部分可以被考虑用作内部RAM。内部RAM又被分割成通用寄存器组（00H~1FH）、可位寻址RAM（20H~2FH）和通用RAM（30H~7FH）。下面分别讨论每一段内部存储器空间。

2.5.1 通用RAM

虽然图2-7只画出了从地址30H~7FH之间80B的通用RAM，但实际上，地址00H~2FH之间的48B的情况与此类似（这些空间还有其他用途，将在后文中讨论）。

24

通用RAM中的任意地址都可以用直接寻址模式或间接寻址模式访问。例如，若要读取内部RAM地址5FH中的内容到累加器中，可以使用下面的指令实现：

```
MOV    A, 5FH
```

该指令传送一个字节的数据，以直接寻址的方式指定被传送数据在存储器中的“源地址”（即5FH）。操作码中还隐含指定了数据的目的地址，即累加器A。（注：寻址

模式将在第3章中详细讨论。）

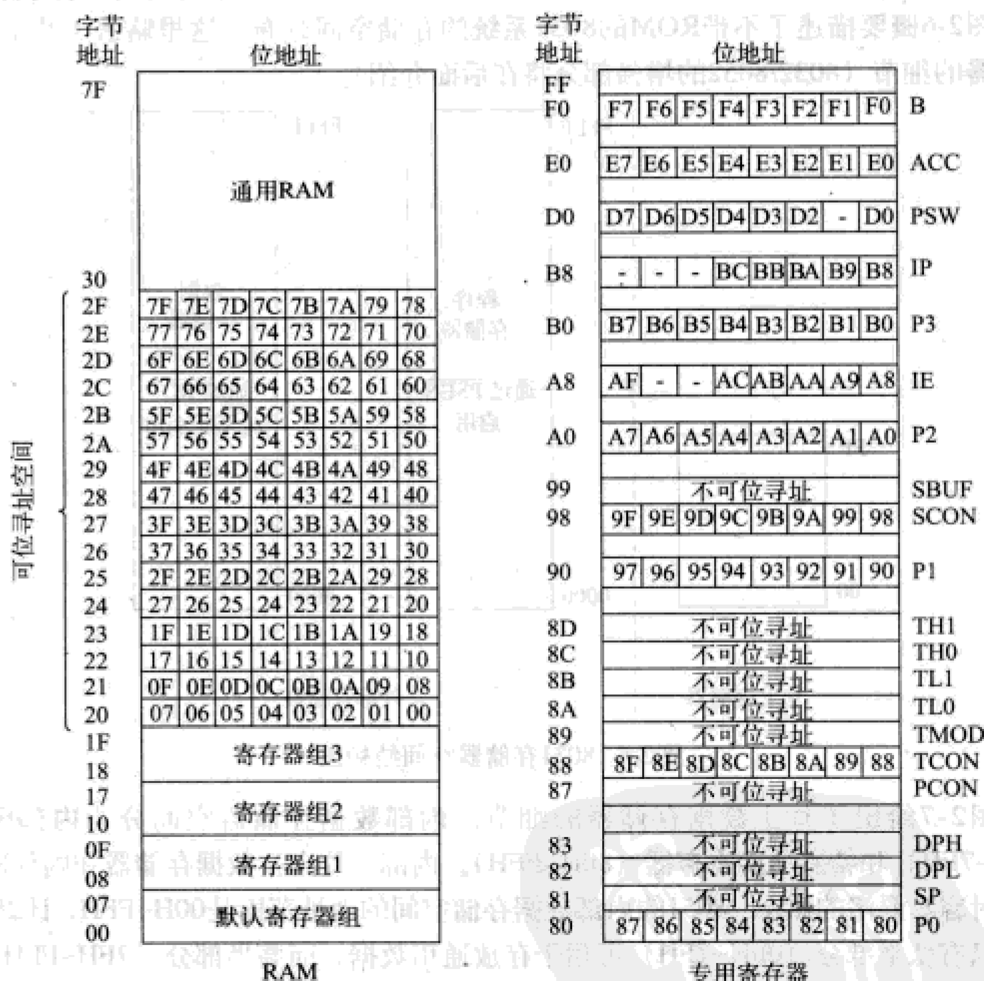


图2-7 8051片上数据存储器结构

内部RAM也可以通过寄存器R0或R1以间接寻址方式访问，例如，下面两条指令执行的结果与上面的一条指令相同：

```
MOV    R0, #5FH
MOV    A, @R0
```

其中，第1条指令以立即数寻址方式将立即数5FH送入寄存器R0，第2条指令的以间接寻址方式将R0内容所指向地址存储的数据送到累加器A中。

25

2.5.2 可位寻址RAM

8051具有210个可位寻址的位置，其中128个位于内部RAM字节地址为20H~2FH的存储空间里，其他的位于部分特殊功能寄存器中（随后讨论）。

允许软件以“位”为单位访问存储器是大多数微控制器的一个非常有用的特性，

仅通过一条指令，就可以实现对相应的位进行置位、清除、与、或等操作。没有位寻址能力的微处理器需要"读－改－写"的指令组合才能完成相同的功能。而且，8051的I/O端口可以位寻址，简化了单个"位"输入/输出时的软件接口。

26

字节地址为20H~2FH的位置包括128个可位寻址的通用存储空间（8位/字节×16字节＝128位）。选择不同的指令，既可以通过字节地址也可以通过位地址来访问这些空间。例如，欲将位地址67H处的内容设置为1，可使用下面的指令实现：

```
SETB    67H
```

参考图2-7，"位地址67H"对应的"位"是"字节地址2CH"所指字节的最高有效位。上面的指令对该字节的其他位没有任何影响。大多数微处理器（没有位寻址功能）通过下面的操作达到相同的目的：

```
MOV   A, 2CH                  ;READ ENTIRE BYTE
ORL   A,#10000000B            ;SET MOST-SIGNIFICANT BIT
MOV   2CH,A                   ;WRITE BACK ENTIRE BYTE
```

例2-1　用什么指令可以将字节地址25H的第3位置1？

答案： SETB　　2BH

讨论： 字节地址25H对应的数据单元位于内部存储器的可位寻址空间内（见图2-7）。从第0位开始，该字节各个位的位地址分别为28H、29H等，所以第3位的位地址是2BH。

2.5.3 寄存器组

内部存储器底部的32B空间是通用寄存器组。8051的指令集支持R0~R7共8个寄存器。默认情况下（系统复位后），这些寄存器位于地址00H~07H处，也就是第0组。因此，下面这条指令将地址05H的内容读到累加器中：

```
MOV   A, R5
```

使用了寄存器寻址方式，指令长度为1B。当然，同样的操作也可通过直接寻址方式完成，指令长度为2B，其中第2个字节为直接地址：

```
MOV   A, 05H
```

经过上述比较可知，完成相同的操作，使用寄存器（R0~R7）寻址的指令比使用直接寻址的指令要短。所以，需要频繁使用的数据应该选择寄存器寻址方式来访问。

改变程序状态字PSW（将在后面讨论）中的工作寄存器组选择位（RS1，RS0）可以改变当前工作寄存器组。假设当前工作寄存器组是第3组，下面的指令将把累加器的内容写入到地址18H中：

```
MOV   R0, A
```

27

"寄存器组"这个概念使软件的不同部分可以占有一组私有寄存器，不受其他

部分的影响，因而可快速有效地进行“上下文切换”。

例2-2 寄存器组3中寄存器5的地址是多少？

答案：1DH

讨论：寄存器组3位于内部存储器空间的地址是18H~1FH（如图2-7所示），R0的地址是18H，R1在19H，依次类推。因此，寄存器5（R5）的地址是1DH。

2.6 特殊功能寄存器

大多数微处理器访问内部寄存器时，寄存器的地址是隐含在指令中的。例如，在6809微处理器中指令“INC A”使累加器A的内容加1。操作的具体含义已经包含在指令操作码中。8051微控制器也采用类似的方法来访问寄存器，同样，指令“INC A”在8051上也完成和微处理器6809相同的操作。

8051的内部寄存器被配置为片上RAM的一部分，因此，每个寄存器都有一个地址①。对于8051来说，这样的安排是非常合理的，因为8051有很多寄存器。除R0~R7以外，还有位于内部RAM顶部的21个专用寄存器，地址范围是80H~FFH（见图2-7和附录D）。注意：从80H到FFH有128B，其中的大多数并未定义，只定义了21个特殊功能寄存器（8032/8052是26个）。

此前已经提到，可以以隐含的方式来访问累加器（A或ACC），但是大多数特殊功能寄存器只能使用直接寻址方式访问。从图2-7可以看出，有些特殊功能寄存器既具有字节寻址能力又具有位寻址能力。程序设计者在使用这些寄存器的时候，要根据具体问题选择位寻址或字节寻址。例如，指令

```
SETB  0E0H
```

设置累加器的第0位，其他位不变。容易混淆的是，E0H既是整个累加器的字节地址又是累加器最低有效位的位地址。但是SETB指令的操作对象是位而非字节，因此只对位地址指定的空间有效。请注意，特殊功能寄存器中的可寻址位有高5位地址与其所属特殊功能寄存器的高5位地址相匹配。例如端口1的字节地址是90H或10010000B，端口1的位地址是从90H~97H，写成通用表示即10010xxxB。

28

例2-3 用什么指令可以置累加器B的最高有效位为1，同时不影响其他位？

答案：SETB 0F7H

讨论：累加器B的地址是F0H，属于特殊功能寄存器（见图2-7），可位寻址。其位0的地址是F0H，位1的地址是F1H，依次类推，位7的地址为F7H。

① 程序计数器和指令寄存器除外，因为很少对二者直接操作，所以将其放到片上RAM没有意义。

下面将详细讨论程序状态字（PSW），然后简要介绍其他的特殊功能寄存器。更详细的讨论将在后面的章节中进行。

2.6.1 程序状态字

程序状态字（PSW）位于地址D0H，每个状态位的定义如表2-3所示。下面分别讨论每一位的含义。

表2-3 PSW寄存器

位	符号	地址	位描述
PSW.7	CY	D7H	进位标志
PSW.6	AC	D6H	辅助进位标志
PSW.5	F0	D5H	标志0
PSW.4	RS1	D4H	寄存器组选择位1
PSW.3	RS0	D3H	寄存器组选择位0 00＝寄存器组0，地址00H~07H 01＝寄存器组1，地址08H~0FH 10＝寄存器组2，地址10H~17H 11＝寄存器组3，地址18H~1FH
PSW.2	OV	D2H	溢出标志
PSW.1	–	D1H	保留
PSW.0	P	D0H	奇偶标志

1. 进位标志

标志（C或CY）有两个用途。在进行算术运算情况下，如果执行加法时第7位有进位或执行减法时第7位有借位，则将标志位C置1。例如，若累加器内容为FFH，则指令

```
ADD    A, #1
```

使累加器中内容变为00H，同时PSW中的进位标志C被置1。

例2-4 执行下面的指令序列之后进位标志位和累加器中的内容发生什么变化？

```
MOV R5, #55H
MOV A, #0AAH
ADD A, R5
```

答案：C＝0，ACC＝FFH

讨论：第3条指令执行的二进制加法如下：

```
  0 1 0 1 0 1 0 1     (R5=55H)
+ 1 0 1 0 1 0 1 0     (ACC=AAH)
  1 1 1 1 1 1 1 1     (累加器中的结果为ACC=FFH)
```

在加法过程中，最高有效位（位7）没有产生进位，因此进位标志为0。累加器的最终结构为FFH＝255_{10}。

进位标志还可以作为“布尔累加器”使用，在布尔指令对位进行逻辑操作时可以把它当作1位的寄存器使用。例如，下面的指令将位地址25H处的内容与进位标志做逻辑与运算，并把运算结果送回到进位标志中：

```
ANL   C, 25H
```

2. 辅助进位标志

当进行BCD（Binary-Coded-Decimal，二进制编码的十进制数）码的加减法时，如果位3向位4有进位或借位，或者低4位的值在0AH~0FH之间时，辅助进位标志（AC）被置1，否则被清零。执行BCD码加法后，必须用DA A指令（十进制调整指令）对运算结果进行调整。

例2-5　执行下面的指令序列后，辅助进位标志的状态如何？这时累加器中的内容是什么？

```
MOV      R5, #1
MOV      A, #9
ADD      A, R5
```

答案：AC＝1，A＝0AH

讨论：第3条指令执行的二进制加法如下

```
             1
  00000001        (R5 = 01H)
+ 00001001        (ACC = 09H)
  00001010        (累加器中的结果为ACC = 0AH)
```

虽然在二进制加法执行过程中没有产生进位，但是结果的低4位是1010B＝AH，大于9_{10}，因此辅助进位标志被置1。如果加法指令紧随一条十进制调整指令（DA A），那么累加器A中的最终结果是00010000B＝10H。在BCD码中，10H＝10_{10}，这正是（$9_{10}+1_{10}$）的正确结果。

30

3. 标志0

标志0（F0）是通用标志位，允许用户定义使用。

4. 工作寄存器组选择位

工作寄存器组选择位（RS0和RS1）用于选择当前工作寄存器组。RS0和RS1在系统复位后都被清零，可以通过软件根据需要改变RS0和RS1的内容。例如，下面的3条指令选择寄存器组3为当前工作寄存器组，然后将寄存器R7中的内容（字节地址1FH）送入累加器A：

```
SETB    RS1
SETB    RS0
MOV     A, R7
```

当汇编上面的指令时，符号“RS1”和“RS0”由它们的位地址代替，指令SETB RS1就等效于SETB 0D4H。

例2-6 假设当前工作寄存器组未知，写出一段指令，使寄存器组2成为当前工作寄存器组。

答案：
```
SETB    RS1
CLR     RS0
```

讨论： 指令的执行结果是使两个工作寄存器选择位中的内容变为$10_2=2_{10}$，寄存器组2成为当前工作寄存器组。如果CPU复位后，RS1和RS0一直未被改变过，那么第2条指令可以省去。但是，在这个题目中工作寄存器选择位的状态未知，因此，必须明确地对RS1和RS0进行相应设置。

5. 溢出标志

执行加减法运算时，如果发生运算溢出，则溢出（OV）标志位被置1，否则被清零。在对有符号数进行加减运算后，软件可以通过检查溢出标志的状态得知计算结果是否在正确的范围之内。如果进行的是无符号数运算，溢出标志无效。但如果是有符号数加法，出现大于+127或小于−128的结果将使溢出标志位被置1。例如，执行下面的加法将发生溢出并使得OV被置1：

```
十六进制：     0F          十进制：     15
            + 7F                     + 127
              8E                       142
```

作为一个有符号数，8EH代表是−114，显然不是正确的计算结果142。因此，OV被置为1。

例2-7 执行下面的指令后溢出标志的状态如何？累加器的内容是什么？

```
MOV R7, #0FFH
MOV A, #0FH
ADD A, R7
```

答案： OV＝0，ACC＝0EH

讨论： R7赋值为FFH，代表的有符号数是−110；累加器A赋值为0FH，即1510。加法的结果是15＋（−1）＝14＝0EH。14在8位有符号数的表示范围之内（−128到+127），因此不会发生溢出，OV为0（注意：进位标志C被置为1，因为位7有进位）。

6. 奇偶标志

每个机器周期之后，奇偶标志（P）都会根据累加器A中内容被置1或清零，原则是使得累加器和奇偶标志位P中的1的总个数始终为偶数，即所谓的偶数奇偶校验。例如：如果累加器中的内容为10101101B，那么P的内容是1（共有6个1，即1的个数为偶数）。奇偶标志通常与串行端口通信程序一起使用，在发送数据之前增加奇偶标志的内容或在接受数据后进行奇偶校验。

例2-8 执行下面的指令后奇偶标志的状态如何？

```
MOV    A, #55H
```

答案： P=0

讨论： 55H转化为二进制是01010101B，其中4位为1，4是偶数，因此P被设置为0。这样累加器和奇偶标志位中的1的数目总和为偶数，满足偶数奇偶校验规则。

2.6.2 寄存器B

寄存器B（或累加器B）的地址是F0H，在乘法和除法运算中与累加器A配合使用。MUL AB指令把累加器A和寄存器B中的8位无符号数相乘，所得的16位乘积的低字节存放在A中，高字节存放在B中。DIV AB指令用B除以A，整数商存放在A中，余数存放在B中。寄存器B还可以用作通用暂存寄存器。寄存器B可位寻址，对应的位地址为F0H~F7H。

32

2.6.3 栈指针

栈指针（SP）是一个8位寄存器，地址是81H。栈指针中存放的是当前栈顶部数据的地址。栈操作包括将数据“压入”栈和将数据“弹出”栈。数据进栈之前，SP先加1，然后再将数据写入当前SP所对应的存储单元；数据出栈时，先读出当前SP所指存储器单元中的数据，然后SP减1。8051的栈建立在内部数据存储器的间接寻址空间里，8031/8051是片上RAM的前128B，8031/8052是片上RAM的所有256B。

要从地址60H开始设立栈，可以用下面的指令：

```
MOV    SP, #5FH
```

对于8031/8051而言，上述指令将使得栈的大小限制在32B以内，因为可使用的片上RAM的最高地址是7FH。该条指令之所以设置SP的初始值为5FH，因为在第1次压栈操作之前，SP先要加1增加到60H。

例2-9 使用什么指令初始化8052的栈指针，可以在其内部数据存储器顶部建立48B的栈？

答案：
```
MOV    SP, #0CFH
```

讨论： 8052有256B的内部存储器，其顶部的48B为D0H~FFH。由于在第1次数据进栈前，SP先要加1，因此SP的初始值应比栈底部少1，为CFH。

在系统复位后，设计人员可以保留栈指针的初始值，不必重新赋值。8051栈指针的初始值是07H，与8048保持兼容，因此栈底地址为08H，也就是说第1个被压栈的数据存储到了08H单元。如果应用软件未重设栈指针的初始值，那么寄存器组1

（也许还有寄存器组2和3）就不能使用，因为这个存储器区域已经被栈占据。

PUSH和POP指令能够直接访问栈以临时性地存取数据，子例程调用指令（ACALL、LCALL）和返回指令（RET、RETI）则间接访问栈以保存并取回程序计数器的内容。

2.6.4 数据指针

数据指针（DPTR）是一个用于访问外部程序或数据存储器的16位寄存器，地址是82H（DPL，低字节）和83H（DPH，高字节）。下面的3条指令将55H写入外部RAM地址为1000H的存储单元中：

```
MOV    A, #55H
MOV    DPTR, #1000H
MOVX   @DPTR, A
```


其中第1条指令以立即数寻址方式将常数55H传送到累加器。第2条指令也利用立即数寻址方式将16位的地址常数1000H传送到数据指针。第3条指令以间接寻址方式将A中的数据（55H）传送到DPTR所指定的外部RAM的相应存储单元（地址为1000H）。“MOVX”助记符中的“X”表示该传送指令的操作对象是外部数据存储器。

2.6.5 端口寄存器

8051的I/O端口中，端口0的地址为80H，端口1的地址为90H，端口2的地址为A0H，端口3的地址为B0H。如果扩展了外部存储器或使用了8051的其他特殊功能（中断、串行通信等），那么端口0、2和3就无法作为通用I/O使用了。但是，在任何情况下，P1.2~P1.7都可作为通用I/O口使用。

所有端口都可以位寻址，这为8051提供了强大的接口功能。例如，如果电机通过电磁铁和晶体管连接在端口P1.7 上，那么仅用1条指令就可以控制电机的开关：

```
SETB   P1.7
```

可以启动电机，而

```
CLR    P1.7
```

可关闭电机。

上面的指令使用点操作符表示可位寻址字节中的某一位，汇编器会做相应的转换。下面两条指令实现的功能是完全相同的：

```
CLR    P1.7
CLR    97H
```

汇编器预定义符号（如P1）将在第7章中详细讨论。

再看另一个例子，在微控制器与设备的接口中定义1个位为BUSY，当设备忙时将其置1，设备就绪时清零。如果把P1的第5位作为BUSY位，下面的循环就可以用于等待设备就绪：

```
WAIT: JB    P1.5, WAIT
```

这条指令的含义是："如果P1.5被置1，则跳转到标号WAIT"。换句话说就是："跳转回去，再次检查"。

2.6.6 定时器寄存器

8051有两个16位定时器/计数器，用于时间间隔定时或事件计数。定时器0的地址为8AH（TL0，低字节）和8CH（TH0，高字节），定时器1的地址为8BH（TL1，低字节）和8DH（TH1，高字节）。定时器操作由位于地址89H的定时器模式寄存器（TMOD）和地址88H的定时器控制寄存器（TCON）决定，其中TCON可位寻址。关于定时器的内容将在第4章中详细讨论。

2.6.7 串行端口寄存器

8051有一个串行端口用于与终端、调制解调器等串行设备通信，或与其他具有串行接口的芯片（A/D转换器、移位寄存器、非易失性RAM等）通信。串行数据缓冲寄存器（SBUF）占据地址99H，用于保存待发送的数据和已接收的数据。通信过程中，将要发送的数据先写入SBUF中，接收到的数据要从SBUF中读取。通过改变串行端口控制寄存器（SCON）的内容可以选择不同的串行端口工作模式，SCON的地址为98H，且可位寻址。关于串行端口操作的详细内容将在第5章讨论。

34

2.6.8 中断寄存器

8051的中断系统有5个中断源，2个中断优先级。系统复位后所有的中断都被禁用，只有修改位于地址A8H的中断启用寄存器（IE）才能允许中断。中断优先级需要通过修改位于B8H的中断优先级寄存器（IP）来进行设置。这两个寄存器都可位寻址。关于中断的内容将在第6章中讨论。

2.6.9 电源控制寄存器

电源控制寄存器（PCON）的地址为87H，其每一位有不同的控制功能，见表2-4。

表2-4 PCON寄存器简表

位	符 号	描 述
7	SMOD	波特率倍增位；如果SMOD被置1，且串行端口工作在模式1、2或3时，那么波特率将提高1倍
6	—	未定义
5	—	未定义
4	—	未定义
3	GF1	通用标志位1

（续）

位	符 号	描 述
2	GF0	通用标志位0
1*	PD	掉电模式位，置1将使8051进入掉电工作模式，只有复位才能退出此工作模式
0*	IDL	待机模式位，置1将使8051进入待机工作模式，中断或系统复位可使8051退出此工作模式

* 仅CMOS版本有效

当串行端口工作在模式1、2和3时，将SMOD置位可使串行端口通信的波特率变为原来的2倍。PCON的位6、5、4没有定义。位3和位2是通用标志位，由用户定义使用。

早期的所有MCS-51™系列产品都有掉电模式位（PD）和待机模式位（IDL）这两个电源控制位，但现在只有CMOS工艺的芯片提供这两个模式位。PCON寄存器不可位寻址。

1. 待机模式

执行完将IDL位置1的指令后，8051进入待机模式。在待机模式下，8051不再向CPU提供内部时钟信号，但是仍向中断、定时器和串行端口提供。CPU保持内部状态不变，所有寄存器中的内容保持不变，所有端口引脚的逻辑电平也维持不变，ALE和 $\overline{\text{PSEN}}$ 维持高电平。

待机模式可以通过触发中断或者重启系统来终止，两个条件都将IDL位清零。

2. 掉电模式

执行完将PD置1的指令后，8051进入掉电模式。在掉电模式下，片上振荡器将停止工作，所有功能停止，所有片上RAM中的内容保持不变，端口引脚的逻辑电平也维持不变，ALE和 $\overline{\text{PSEN}}$ 维持低电平。退出掉电模式的唯一方法是系统复位。

处于掉电模式期间，允许 V_{cc} 最低降到2V。这里需要注意，只有系统已经进入掉电模式后才能降低 V_{cc}。离开掉电模式时，在RST信号回到低电平之前，V_{cc} 需恢复到5V并至少保持10个振荡周期。

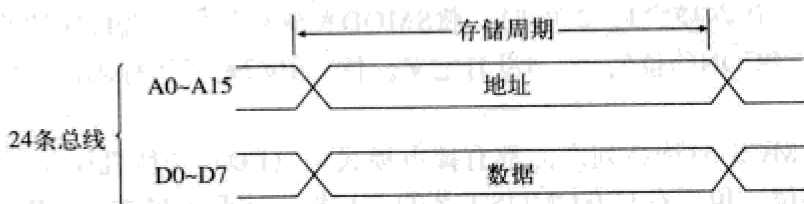
2.7 外部存储器

控制器的片上资源有限，为避免由此造成的设计应用上的潜在限制，微控制器的扩展能力就显得格外重要。任何资源（存储器、I/O等）若是需要扩展，则要求微控制器应具备相应的能力。MCS-51™架构提供了64KB外部程序存储空间和64KB外部数据存储器空间的扩展能力，可根据需要增加外部ROM和RAM。还可增加外围接口芯片来扩展它的I/O能力，扩展的I/O通道以存储器映像的方式成为外部数据存储器空间的一部分。

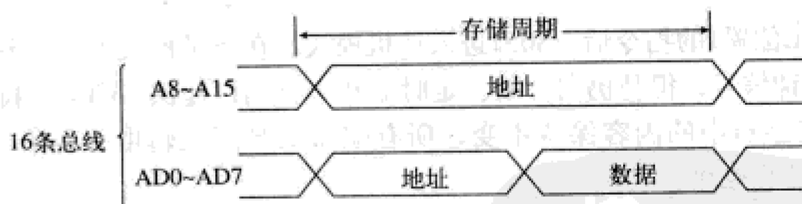
当使用外部存储器时，端口0不再作为I/O口使用，而是作为复用的地址总线（A0~A7）和数据总线（D0~D7），在每个外部存储器存取周期的前半个周期利用ALE信号锁存地址信息的低字节。端口2通常（并不总是）用作高位地址总线。

36

在讨论8051的地址总线和数据总线复用的细节之前，先看以下总线复用的一般情况。如图2-8所示，不作复用的总线排列方式需要16条专用地址总线和8条专用数据总线，共计需要24个引脚。在总线复用的方式下，数据总线和地址总线的低8位共用8条总线，另外8条总线用作地址总线的高位地址，共需要16个引脚。在40个引脚的DIP封装芯片上，节省下来的引脚就可以为其他功能所用。



(a) 非复用状态（24条总线）



(b) 复用状态（16条总线）

图2-8　地址总线和数据总线的复用

总线复用的工作方式是：在每个存取周期的前半个周期，端口0传输地址信息的低字节，用ALE信号锁存，在该存储周期中，地址信息一直保存在74HC373（或同类型芯片）锁存器中，在后半个存储器周期，端口0作为数据总线使用，而且究竟是读还是写要由相关的指令确定。

2.7.1　外部程序存储器的访问

外部程序存储器是只读存储器，启用信号是 $\overline{\text{PSEN}}$ 。使用外部EPROM时，端口0和端口2都不能用作通用I/O端口。外部EPROM与8051的连接方式如图2-9所示。

1个8051机器周期包含12个振荡周期。如果片上振荡器由12MHz的晶振激励，那么每个机器周期的持续时间是1μs。在1个典型的机器周期中，ALE信号输出2个脉冲，CPU从程序存储器中读取2个字节（如果当前读取的指令只有1个字节，那么放弃第2个字节）。执行这个操作（获取指令）的时序如图2-10所示。

37

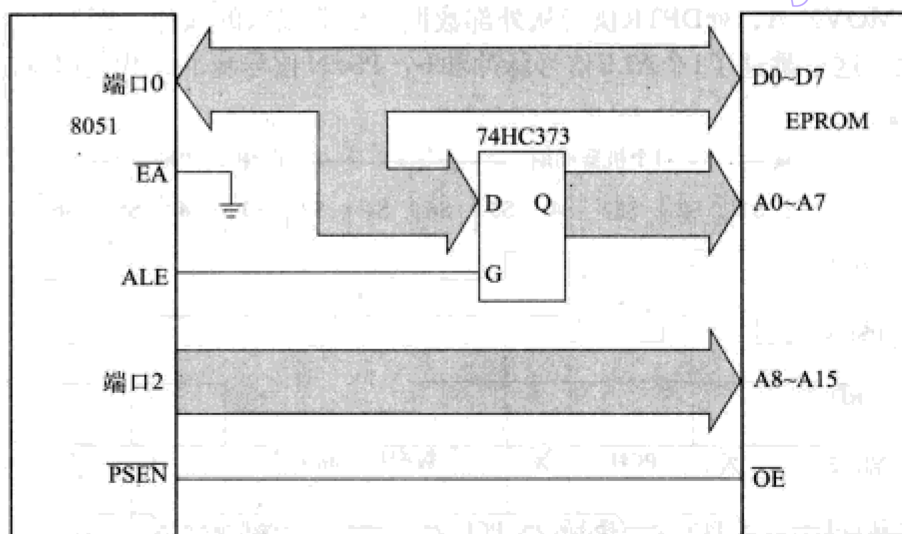


图2-9 访问外部程序存储器

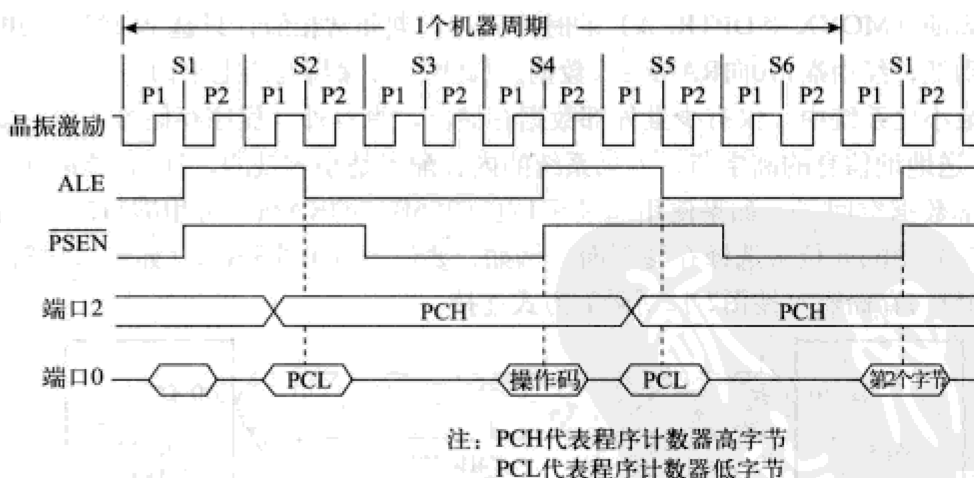


图2-10 读取外部程序存储器的时序图

2.7.2 外部数据存储器的访问

外部数据存储器是可读/可写存储器，利用 $\overline{RD}$ 和 $\overline{WR}$ 信号（即引脚P3.7和P3.6的第二功能）选通。访问外部数据存储器的唯一方法是利用指令MOVX，另外需要以数据指针DPTR、R1或R0作为地址寄存器。

RAM与8051的连接方式与EPROM基本相同。不同的是需要将 $\overline{RD}$ 信号线与RAM的 $\overline{OE}$（输出启用）信号线相连，$\overline{WR}$ 信号线与RAM的 $\overline{W}$（写入）信号线相连。地址和数据总线的连接方法与EPROM相同。利用端口0和端口2，8051最多可扩展64KB的外部RAM。

指令MOVX A，@DPTR执行从外部数据存储器读取的操作，图2-11是其时序图。注意，这里跳过了1个ALE信号脉冲和1个 $\overline{PSEN}$ 信号脉冲，代之以 $\overline{RD}$ 信号选通RAM[①]。

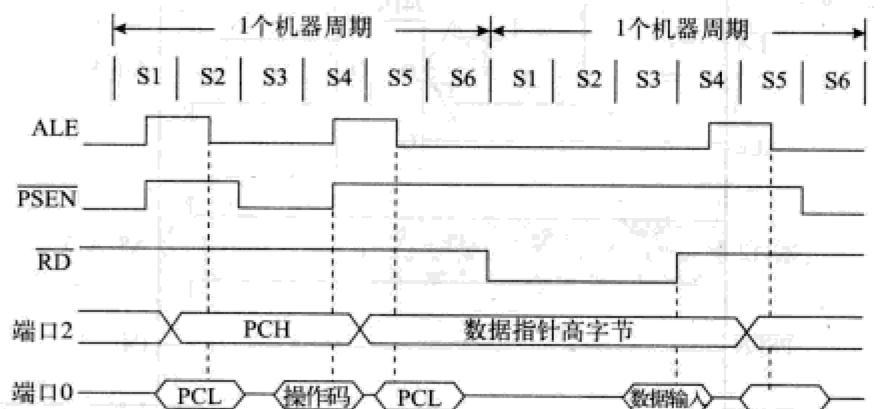


图2-11 MOVX指令时序

写周期（MOVX @DPTR, A）的时序同读周期非常相似，只是 $\overline{WR}$ 线上的电平由高变到低，经由端口0向RAM写入数据。（此时 $\overline{RD}$ 保持为高电平。）

在最小化系统中，仅有少量外部数据存储器，没有外部程序存储器，不需要端口2来传送地址信息的高字节，如果系统的内存配置是小容量的，那么8位地址足以访问外部数据空间了。如果使用超过了1页（256B）的RAM，可用端口2（或其他未用端口）中的几位来选择存储页面。例如，要使用1KB的RAM（如4个256B的页面），8051与存储器可按图2-12所示的方式连接。

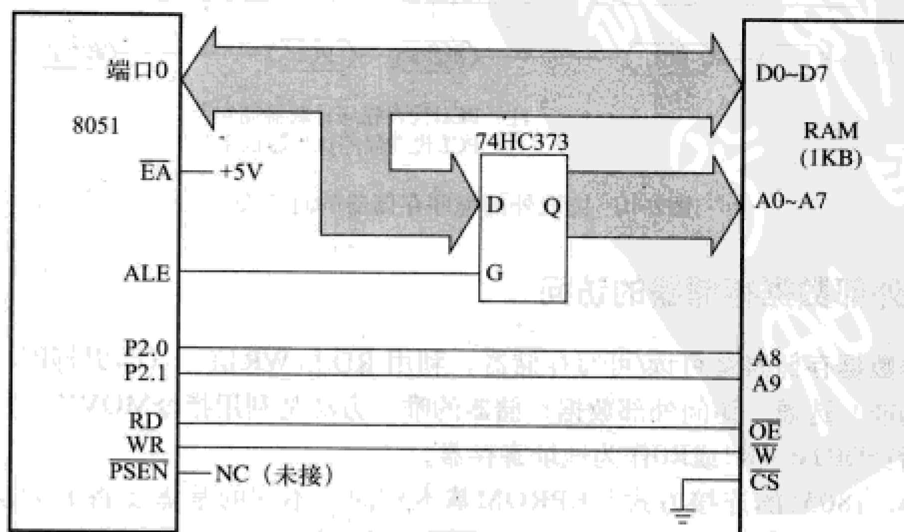


图2-12 8051和1KB RAM的接口

① 如果MOVX指令（访问外部RAM）在某设计中没有使用，那么ALE脉冲可作为频率为晶振1/6的时钟信号使用。

在这种情况下，欲使用MOVX指令读写数据，必须先初始化端口2的第0位和第1位，选择要读写的存储器页面，然后指令才能在页内空间进行读写操作。例如，设P2.0＝P2.1＝0，那么下面的指令可以将外部RAM地址0050H中的内容读到累加器中：

38
~
39

```
MOV        R0, #50H
MOVX       A, @R0
```

要读取该RAM中最后1个地址（03FF）的内容，必须将两个页面选择位置1。可以执行下面的指令来实现：

```
SETB          P2.0
SETB          P2.1
MOV           R0, #0FFH
MOVX          A,  @R0
```

这样设计的一个特点是端口2的位2到位7无需作为地址位使用，可以作为通用的输入/输出端口。如果用DPTR作为地址寄存器，那么，端口2就必须被全部用于传输地址信息的高字节。

40

2.7.3 地址译码

如果8051连接了多片EPROM或RAM，则需要进行地址译码。8051地址译码的过程与大多数微处理器相似。例如，如果使用8KB的EPROM或RAM芯片，必须对地址信息进行译码，以确定是哪个存储器芯片的8KB范围：0000H~1FFFH，2000H~3FFFH等。

典型的译码芯片（如74HC138）的连接方式是将其输出端与存储器的片选（$\overline{CS}$）的输入相连。图2-13描述的是一个包含了多片2764 EPROM（8KB）和6264 RAM（8KB）存储芯片的系统。需要注意的是，由于启用信号的不同（程序存储器使用$\overline{PSEN}$，数据存储器使用$\overline{RD}$和$\overline{WR}$信号），8051最大可以分别使用64KB EPROM和64KB RAM。

2.7.4 外部程序存储空间和数据空间的重叠

程序存储器是只读存储器，这给8051的软件开发带来了困难。如果软件只能存储在只读的程序存储器中以供系统调用执行，那如何将软件写入存储器进行调试呢？通常使用的方法是将外部程序和数据存储器空间重叠。$\overline{PSEN}$用于控制读取程序存储器，$\overline{RD}$用于控制读取数据存储器，如果将RAM的$\overline{OE}$信号同$\overline{PSEN}$和$\overline{RD}$的“逻辑与”（反相输入或非）相连，RAM就可以同时作为程序和数据存储器，连接电路如图2-14所示，允许RAM芯片作为数据存储器写入，并可作为数据或程序存储器读取。这样，先将RAM作为数据存储器使用，将程序写入，再作为程序存储器，从中读取程序并执行。

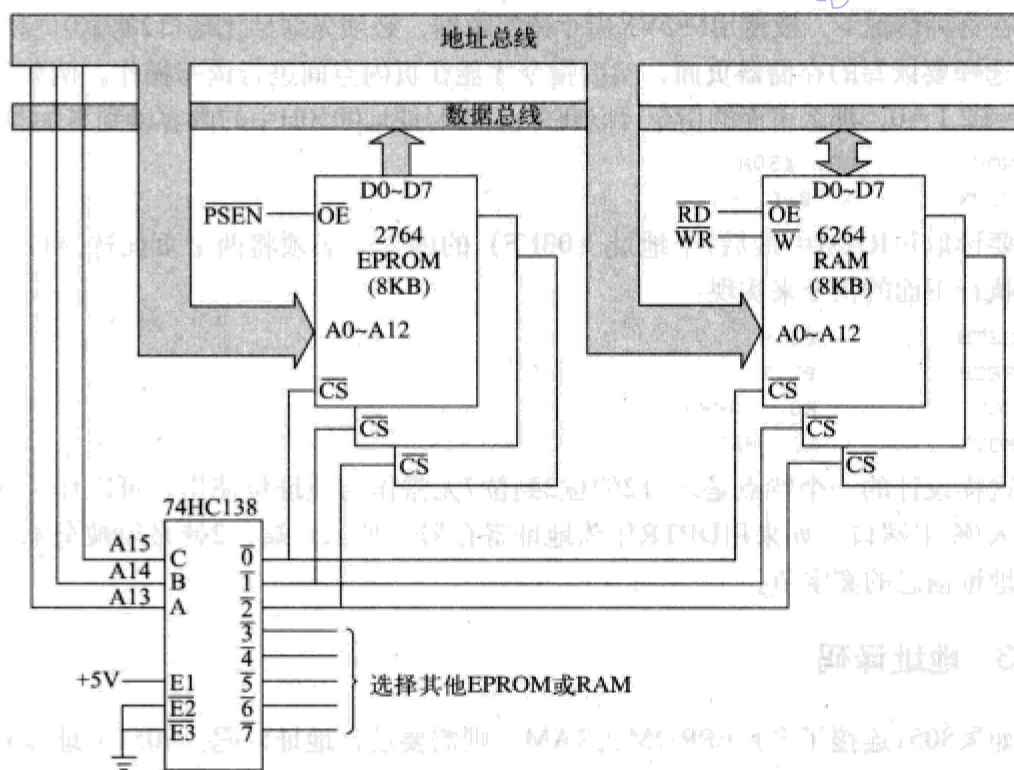


图2-13 地址解码

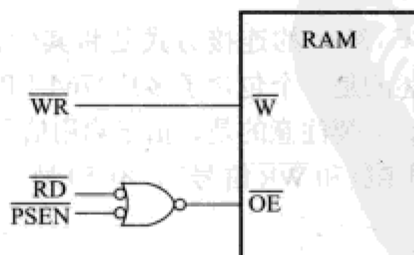


图2-14 外部存储器程序与数据空间的重叠

41

2.8 8032/8052的增强功能

8032/8052微控制器（及其CMOS型和EPROM型芯片）与8031/8051相比增加了两个功能。第1个是增加了128B的片上RAM，地址为80H~FFH。为了不和特殊功能寄存器（与增加的RAM地址相同）产生地址冲突，新增加的1/8KB RAM只能以间接寻址方式被访问。指令

```
MOV     A, 0F0H
```

在所有MCS-51™系列微控制器上都是将寄存器B的内容送入累加器A中。而指令

```
MOV        R0, #0F0H
MOV        A, @R0
```

在8032/8052芯片上执行的结果是将内部RAM中的F0H的内容读入累加器A，而在8031/8051芯片上则未定义。8032/8052的内部存储器结构如图2-15所示。

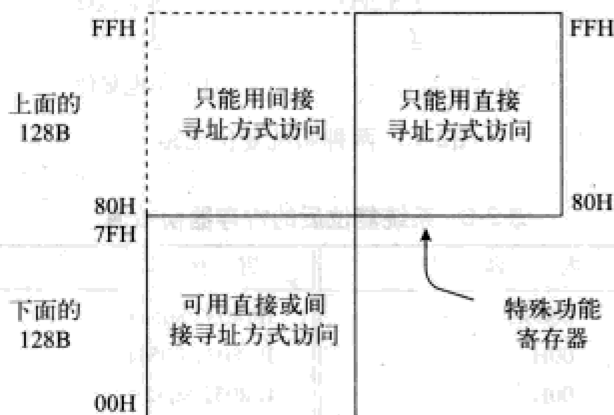


图2-15 8032/8052的存储空间

8032/8052的第2个增强功能是增加了一个16位定时器——定时器2，该定时器的控制通过5个新增的特殊功能寄存器实现。表2-5给出了这5个特殊功能寄存器的简要说明，更多的细节将在第4章介绍。

表2-5 定时器2的寄存器

寄存器	地址	描述	可否位寻址
T2CON	C8H	控制寄存器	是
RCAP2L	CAH	捕获寄存器低字节	否
RCAP2H	CBH	捕获寄存器高字节	否
TL2	CCH	定时器2寄存器低字节	否
TH2	CDH	定时器2寄存器高字节	否

2.9 复位操作

将RST信号变成高电平并维持至少2个机器周期然后再变回低电平，8051就会执行复位操作，然后把RST恢复为低电平。RST信号可以利用按键来手动产生，也可在上电过程中利用RC网络产生。图2-16是两种不同复位方式的电路图。

复位后8051所有的寄存器状态都在表2-6中列出。其中最重要的寄存器可能是程序计数器，系统复位后其初始值为0000H。当RST恢复为低电平后，系统总是从程序存储器的起始处开始执行，起始地址即0000H。复位操作不影响片上RAM的内容。

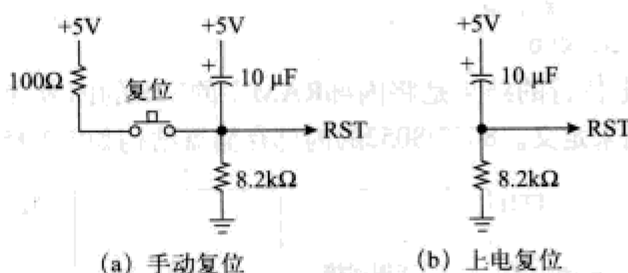


图2-16 两种系统复位电路

表2-6 系统复位后的寄存器初始值

寄存器	内容	寄存器	内容
程序计数器	0000H	IP(8032/8052)	XX000000B
累加器	00H	IE(8031/8051)	0XX00000B
寄存器B	00H	E(8032/8052)	0X000000B
PSW	00H	定时器寄存器	00H
SP	07H	SCON	00H
DPTR	0000H	SBUF	00H
端口0~3	FFH	PCON(HMOS)	0XXXXXXXB
IP(8031/8051)	XXX00000B	PCON(CMOS)	0XXX0000B

43

小结

本章简要介绍了8051微控制器的硬件体系结构。但是在进行应用开发之前，读者还必须了解8051指令集。下一章将主要介绍8051的指令和寻址模式。本章有意简略了关于定时器、串行端口和特殊功能寄存器的介绍，因为后续章节将专门对此进行详细讨论。

习题

2.1 写出4个8051微控制器制造商的名字（Intel除外）。

2.2 MCS-51™系列的哪种芯片可容纳较大的片上程序并可用于大规模的产品中？

2.3 写出可将字节25H的最低有效位置1的指令。

2.4 写一段指令，对位地址00H和01H中的内容进行或运算，并将结果送到位地址02H中。

2.5 写一段指令，读取端口0的第0位的状态并写入端口3的第0位。

2.6 写一段指令，读取端口0的第0、1位的状态，如果都为1，则向端口3的第0位写1，否则写0。

2.7 写一段指令，读取端口0的第0、1位的状态，如果一位为1且另一位为0，那么向端

口3的第0位写0，否则写1。

2.8 写一段指令，读取端口0的第0、1位的状态，如果任一位为1，向端口3的第0位写0，否则写1。

2.9 用逻辑门表示题2.6~题2.8中完成的操作。

2.10 执行下面的指令后，哪个位的地址被置1？

```
a. MOV     26H, #26H
b. MOV     R0, #26H
   MOV     @R0, #7AH
c. MOV     A, #13H
d. MOV     30H, #55H
   XRL     30H, #0AAH
e. SETB    P1.1
f. MOV     P3, #0CH
```

2.11 下面的指令长为2B，写出与此等效的1B长的指令。

```
MOV   0E0H, #55H
```

2.12 写一段指令，将数值0ABH存入外部RAM地址为9A00H的存储单元。

2.13 8052中定义了多少个特殊功能寄存器？

2.14 8051系统复位后栈指针的初始值是多少？

2.15 写指令设置栈指针，在8031或8032的内部存储器空间的顶部建立64B大小的栈。

2.16 写指令设置栈指针，在8051或8052的内部存储器空间的顶部建立32B大小的栈。

2.17 某子例程可以扩展寄存器R0~R7的应用，在进入该子例程时，它切换寄存器组3为当前工作寄存器组，并在退出子例程时恢复为之前的寄存器组，解释这样的操作如何实现。

2.18 执行下面的各条指令后，当前工作寄存器组分别是哪一组？

```
a. MOV     PSW, #0FDH
b. MOV     PSW, #18H
c. MOV     PSW, #08H
```

2.19 执行下面的各条指令后，当前工作寄存器组分别是哪一组？

```
a. MOV     PSW, #0C8H
b. MOV     PSW, #50H
c. MOV     PSW, #10H
```

2.20 80C31BH-1可以工作在16MHz下，晶振信号从引脚XTAL1和XTAL2输入，在未使用MOVX指令的情况下，ALE信号的频率是多少？

2.21 如果8051的晶振工作在4MHz下，1个机器周期是多长时间？

2.22 如果8051的晶振工作在10MHz下，且软件不读写外部RAM，那么ALE信号的频率是多少？

2.23 如果不读写外部RAM，ALE信号的占空比是多少？（占空比指信号为高电平的时间占一个周期总时间的比例。）

2.24 根据2.8节所述，如果RST引脚保持高电平2个机器周期以上会使8051复位（如附录E所示，RST“高”引脚电平的最小值为2.5V）。



(a) 如果8051的晶振为8MHz，要使系统复位，RST应最少维持多长时间的高电平?

(b) 图2-15a描述了手动复位的RC电路。当复位键被按下时，RST电平为5V，系统进入复位状态。松开复位键后多久8051仍处于复位状态?

2.25 端口引线P1.7可以驱动多少个LS TTL负载?

2.26 给出8051用于选通外部EPROM和外部RAM的控制信号的名称。

2.27 8051内部数据存储器地址25H的最高有效位的位地址是多少?

2.28 8051内部数据存储器地址2FH的第3位的位地址是多少?

2.29 8031片上数据存储器的部分可位寻址空间被用作系统信号的映像，是哪些信号?给出相应的引脚和对应的位地址。

2.30 指出下面的SETB指令操作的位所在的字节地址及其在字节中的位置。

```
a. SETB      0A8H
b. SETB      84H
c. SETB      63H
```

2.31 指出下面的SETB指令操作的位所在的字节地址及其在字节中的位置。

```
a. SETB      37H
b. SETB      77H
c. SETB      0F7H
```

2.32 写出可将累加器A的最低有效位置1而不影响其他位的指令。

2.33 执行下面各指令后，程序状态字的奇偶校验位的状态分别是什么?

```
a. MOV       A, #55H
b. MOV       A, #0F8H
c. MOV       A, #0FFH
```

2.34 执行下面各指令后，程序状态字的奇偶校验位的状态分别是什么?

```
a. CLR       A
b. MOV       A, #03H
c. MOV       A, #0ABH
```

2.35 写一段指令，将寄存器R7中的内容复制到外部RAM地址为100H的存储单元中。

2.36 写一段指令，读取外部RAM地址为08FAH的存储单元中的内容并送入到累加器B中。

2.37 如果系统复位后执行第1条指令是子例程调用指令，那么在跳转到子例程前，程序计数器的内容将被保存在内部RAM中。请问程序计数器的值保存在哪个地址中?

2.38 指令MOV SP, #08FH在8032上执行的结果是什么? 在8031上执行的结果是什么?

2.39 如果8031的程序只用到第0组寄存器，则不需要改变栈指针的值，如果程序要用到全部的4组寄存器，则必须修改栈指针的值，为什么?

2.40 8051的待机模式和掉电模式有什么区别?

2.41 什么指令可以强制8051进入掉电模式?

2.42　说明如何连接8051和2片32KB的静态RAM可以构成64KB外部数据存储空间。

2.43　如果在下述状态下发生系统复位，相应的内容会发生什么变化？

```
A=55H
B=11H
Internal RAM location 30H=33H
SP=00H
```

2.44　解释概念“I/O扩展”。

2.45　存储器映射的I/O和真正的I/O端口有什么区别？

2.46　8051微控制器运行的程序保存在外部ROM中，详细说明其访问外部ROM的控制信号和在启用外部ROM时这些控制信号的逻辑值；另外，说明能够确定第1条指令被成功读取的寄存器（以及该寄存器在系统复位后的初始值）。

2.47　通常的说法是8051有128B的片上数据存储器，但是，参考图2-7的存储器结构图可知，8051的片上有从地址00H到FFH共计256个存储单元，原因是什么？

46
~
48

2.48　通过1个例子解释栈和栈指针的区别。

第3章　指令集概述

3.1　引言

如同每个句子都由单词组成一样，程序是由指令构成的。如果在设计程序时严格按照语法逻辑精心构造指令序列，那么编写出来的程序不但运行速度快、执行效率高，还很具有美感。不同的计算机有不同的指令集；所谓指令集，就是指诸如ADD、MOVE、JUMP等计算机的基本操作的集合。本章引导读者了解MCS-51™的各种寻址模式，通过若干典型程序设计例子的源代码，介绍MCS-51™的指令集。附录A是一张关于所有8051指令的速查表。附录C给出了每条指令的详细介绍。这些附录可以作为后续参考。

本章不讨论编程技术，也不讨论汇编器程序是如何将汇编语言程序（助记符、标号等）转换成机器语言程序（二进制代码）的，这些是第7章的内容。

MCS-51™的指令集为8位控制应用需求做了优化。其中包括了多种寻址模式，可简洁、快速地访问内部RAM，方便计算机对小型数据结构进行操作。指令集包括大量支持单个"位"变量的指令，在需要进行布尔运算的控制及逻辑系统中，这些指令可直接对"位"进行操作。

作为一种典型的8位处理器，8051指令的操作码长度是8位。8位长度的操作码在理论上最多能提供$2^8=256$条指令。在8051中，设置了255条指令，另外还有1条指令未定义。除了操作码之外，某些指令还额外包含1B~2B的操作数，用作数据或地址。总之，在8051中，有139条1B的指令，92条2B的指令，24条3B的指令。在附录B中给出了一张操作码映射表，表中注明了每条指令对应的助记符、指令所包含的字节数以及执行指令所需要的机器周期数。

3.2　寻址模式

指令运行时，如果要对数据进行操作，那么就有一个问题：数据在何处？要回答这个问题，就需要了解8051的寻址模式。8051提供了多种寻址模式，因而这个问题就有多种答案，例如，"在指令的第二个字节"，"在寄存器R4中"，"在直接地址35H对应的内存单元中"，"在数据指针指向的地址的外部数据存储器中"，等等。

寻址模式是每一种计算机的指令集中不可缺少的一部分。寻址模式规定了数据的来源和目的地，对不同的程序指令，来源和目的地的规定也会不同。本节将会讨

论8051的所有寻址模式，且对每种模式都给出了相应的例子。在8051中，共有8种寻址模式（如图3-1所示）：

- ❑ 寄存器寻址
- ❑ 直接寻址
- ❑ 间接寻址
- ❑ 立即寻址
- ❑ 相对寻址
- ❑ 绝对寻址
- ❑ 长寻址
- ❑ 索引寻址

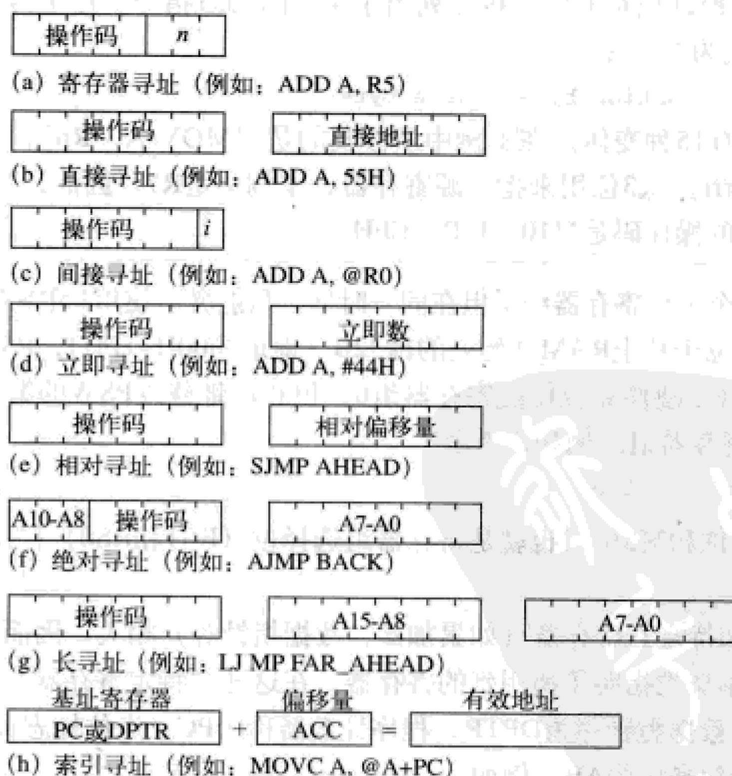


图3-1 8051的各种寻址模式

3.2.1 寄存器寻址

在设计8051程序时，可访问8个“工作寄存器”，编号从R0~R7。如果指令采用寄存器寻址模式，那么指令的操作码的低3位被用来指定这8个工作寄存器的其中之一。这样，指令的功能代码和操作数的地址可组合起来，构成一个简洁的1B指令（请参考图3-1a）。

在8051的汇编语言中，用符号R*n*（*n*的范围为0~7）来代表寄存器。例如，若要把寄存器R7中的数据加到累加器中，可用如下指令：

```
ADD A,R7
```

相对应的操作码是00101111B。高5位的00101表示指令的功能，低3位的111代表寄存器R7。为了确认该操作码正确与否，读者可自行查阅附录C中的指令定义。

例3-1　下面的指令对应的操作码是多少？该指令的功能是什么？

```
MOV  A,R7
```

答案：EFH。该指令把寄存器R7（当前激活的寄存器组中的R7）中的8位数据传送到累加器*A*中。

讨论：附录C按照助记符的字母排序列出了所有的8051指令。用于传送字节数据的指令的一般格式为：

50

```
MOV  destination_byte,source_byte
```

MOV指令有15种变体。在本例中所涉及的是“MOV A，Rn”，对应的二进制操作码是11101rrr。低3位用来指示源寄存器，本例中是R7。因而，用“111”代替“rrr”后，生成的操作码是11101111B＝EFH。

8051包含4个工作寄存器组，但在同一时间，只能激活使用1组寄存器。实际上，这4个寄存器组位于片上RAM开始处的前32B（地址为00H~1FH），PSW的位4和位3决定激活哪个组。硬件复位使能寄存器组0，但相应地修改PSW的第3位和第4位可以选择不同的寄存器组。例如，指令

51

```
MOV  PSW,#00011000B
```

把PSW中的第4位和第3位［也就是寄存器组选择位（RS1和RS0）］设置为1，激活了寄存器组3。

某些指令和特定的寄存器（如累加器、数据指针等）相关，因而不需要用到地址位。操作码本身就指明了所用到的寄存器。在这些“特定寄存器”的指令中，把累加器称为*A*，数据指针称为DPTR，程序计数器称为PC，进位标志称为C，累加器和寄存器*B*组合起来称为AB，例如：

```
INC   DPTR
```

是1B的指令，把16位的数据指针加1。请参阅附录C找出本条指令的操作码。

例3-2　（a）下面指令的操作码是什么？（b）该指令的功能是什么？

```
MUL AB
```

答案：（a）A4H。（b）该指令把累加器*A*中的8位无符号数据和寄存器*B*中的8位无符号数据相乘，16位乘积的低位字节置于累加器中，高位字节置于寄存器*B*中。

3.2.2 直接寻址

直接寻址可访问任何片上变量或硬件寄存器。在指令操作码之后，附有一个字节（直接地址），该字节用来指明待访问的数据单元的地址（请参考图3-1b）。

直接地址的最高位被用来确定应该选择两类片上存储器中的哪一类。如果第7位为0，那么直接地址的范围是从0~127(00H~7FH)，对应片上RAM的低128B。但是，所有的I/O端口寄存器、特殊功能寄存器、控制寄存器或状态寄存器的地址都处于128~255中(80H~FFH)。如果直接地址的第7位为1，那么指令就会访问相应的特殊功能寄存器。例如，端口0和端口1被分配的地址分别是80H和90H。通常而言，无需知道这些寄存器的地址是多少。汇编器允许在程序中使用助记符的简写形式（P0表示端口0，TMOD表示定时器模式寄存器），并且能理解这些简写形式的含义。某些汇编器（比如Intel的ASM51）已经自动包含了预定义符号的定义。其他汇编器可能会用一个单独的源文件来包含这些定义。下面的指令是一个直接寻址的例子：

```
MOV   P1, A
```

把累加器*A*中的数据传送到端口1。汇编器会自行确定端口1（90H）的直接地址，并把其插入到指令的第2个字节处。操作码隐含着源操作数是累加器*A*。具体情况请参阅附录C。该指令的二进制代码如下：

```
10001001——第一个字节（操作码）
10010000——第二个字节（端口1的地址）
```

52

例3-3　如下指令对应的机器语言字节是什么？

```
MOV   SCON,#55H
```

答案：75H，98H，55H。

讨论：该类指令的一般形式为

```
MOV   direct,#data
```

在附录C中，注明了该指令的长度是3B。第1个字节是操作码75H，第2个字节是特殊功能寄存器SCON的直接地址98H（如图2-6所示或参考附录D），第3个字节是立即数55H。

3.2.3 间接寻址

如果一个变量的地址在程序运行的时候被确定计算或者修改，这个变量该如何表示呢？在访问存储器中顺序排列的数据单元或访问RAM中建立起来的表、多精度数据以及字符串的时候，都会遇到这样的问题。此时，无法采用寄存器寻址和直接寻址，因为这两种模式都要求操作数的地址在汇编的时候就能确定下来。

8051为上述问题提供的解决方案是间接寻址。R0和R1被用作“指针”寄存器——R0和R1中的数据表示内部RAM中的一个地址，该地址对应的存储单元中放着指令

所要存取的数据。指令操作码的最低位被用来确定R0为指针还是R1为指针（请参考图3-1c）。

在8051的汇编语言中，间接寻址采用R0或R1前添加“@”符号来表示。例如，假设R1中的数据是40H，内部数据存储器40H单元所包含的数据为55H，那么如下指令：

```
MOV  A,@R1
```

把数据55H传送到累加器。

例3-4 (a) 下面指令的操作码是什么？(b) 该指令的功能是什么？

```
MOV A,@R0
```

答案：(a) E6H。(b) 该指令把内部RAM中的某个数据单元的一个字节的数据传送到累加器中。该数据单元的地址由寄存器R0中的值所确定。

讨论：该类指令的一般格式为：

53

```
MOV  A,@Ri
```

从附录C中可以查到，对应的二进制操作码为1110011i。由于R0被用来作为间接寻址的寄存器，因而用0来代替操作码中的i。操作码是11100110B＝E6H。

在那些需要逐个访问地址连续内存单元的场合，间接寻址至关重要。例如，下面的指令序列把RAM中地址从60H~7FH的各个数据单元清零：

```
        MOV     R0,#60H
LOOP:   MOV     @R0,#0
        INC     R0
        CJNE    R0, #80H, LOOP
        (continue)
```

第1条指令把R0的值初始化为内存块的起始地址值，第2条指令采用间接寻址，把R0指向的数据单元的值置为00H，第3条指令递增R0的值，使其指向下一个地址，最后一条指令测试R0的值，确定是否已经到了内存块的结束处。由于R0的递增在间接存取之后发生，所以R0必须与80H比较，不能与7FH比较，这样才能保证结束处的内存单元（7FH）在循环结束前已经被置零。

3.2.4 立即寻址

如果源操作数是一个常数而非变量（即指令中用到的数据，其值在汇编的时候已经确定），那么该常数可以作为立即数的一位嵌入到指令中，但指令需要添加1个字节用来存放该常数（参考图3-1d）。

在8051的汇编语言中，如果源操作数是立即数，那么在源操作数之前需要加一个“#”符号。源操作数可以是一个数字常数，也可以是一个符号变量，还可以是由常数、符号和运算符组成的算术表达式。汇编器会计算表达式的值并把常数嵌入

到指令中。例如，指令

```
MOV  A,#12
```

把12（0CH）装入累加器A中（汇编器在“12”之后没发现H标志，因而认为该常数是十进制数）。

除了一个例外，所有立即寻址指令的立即数都是8位常数。例外就是，在初始化数据指针的时候，需要一个16位常数。例如

```
MOV  DPTR, #8000H
```

是一个3B指令，该指令把16位的常数8000H装载到数据指针DPTR中。

例3-5　如下指令的二进制形式和十六进制形式的机器语言代码各是什么？

```
ADD  A,#15
```

54

答案：二进制形式为：00100100B，00001111B。

十六进制形式为：24H，0FH。

讨论：该指令的一般格式为

```
ADD  A, #data
```

查阅附录C可知，其对应的操作码是00100100B＝24H，指令的第二个字节是立即数。在上面的例子中，立即数为15_{10}＝00001111B＝0FH。

3.2.5　相对寻址

相对寻址仅在某些特定的跳转指令中用到。相对地址（偏移量）是有符号的8位数据，在执行跳转指令的时候，8051把该数据和程序计数器PC的值相加，所得到的和即为跳转的目标地址，也就是下一条要执行指令的地址。由于相对地址是有符号的8位数据，因而，偏移量的范围为−128~+127。相对偏移量作为一个字节，被添加在指令的操作码之后（参考图3-1e）。

在执行加法操作前，8051会先递增程序计数器的值，使其指向紧跟在跳转指令后的下一条指令，因此，偏移量是相对于指令后的下一条指令而言，而不是相对于跳转指令本身（参考图3-2）。

跳转指令的目的地通常是用标号表示的，汇编器在汇编源程序的时候，会自动计算相对偏移量，因此，一般情况下编程者无需关注此类细节。例如，如果标号THERE代表地址1040H处的指令，如下指令：

```
SJMP  THERE
```

位于内存空间地址1000H和1001H处，汇编器在汇编的时候，计算出相对偏移量为3EH(1002H+3EH＝1040 H)，并会把SJMP指令的第2个字节置为3EH。

55

相对寻址的优点是，采用相对寻址的程序源代码是与位置无关的（因为没有用到“绝对”地址）；其缺点在于跳转的范围有限。



图3-2 计算相对寻址中的偏移量

例3-6 如下指令

```
SJMP 9030H
```

位于内存中地址9000H和9001H处，那么该指令对应的机器语言代码是什么？

答案：80H，2EH。

讨论：该类指令的一般格式为：

```
SJMP relative
```

查阅附录C可知，该指令的长度是2B，第1个字节是操作码10000000B=80H，第2个字节为相对寻址的偏移量，是8位的有符号数。本例中SJMP指令是前向跳转，因而，偏移量是正的。如图3-2所示，源地址是跳转指令之后的那条指令的地址，把偏移量加到源地址上所得到的和就是跳转的目标地址。为了计算出偏移量，如图3-2所示，偏移量可以按照如下算术公式计算出来，而不必画图并计算偏移量：

源地址+偏移量=目的地址

或者

偏移量=目的地址-源地址

在本例中，源地址是9002H（紧接SJMP指令后的下一条指令的地址），目标地址是9030H，因而：

偏移量=9030H-9002H=2EH

例3-7 在内存空间地址0802H和0803H处有一条SJMP指令，其机器语言表达式为80H F6H，那么，该指令跳转的目标地址是多少？

答案：07FAH。

讨论：该跳转指令的偏移量为F6H，这是一个负数，也就是说，8051将会在内存空间中向后跳转。源地址是紧接跳转指令后的下一条指令的地址，在上面例子中，源

地址是0804H。把偏移量加到源地址上，就可以得到目标地址。但这里有一个小诀窍，由于偏移量是负的8位数据，而源地址是16位的数据，因而，进行加法运算前，必须对偏移量进行“符号扩展”，使其变成一个16位的数据FFF6H。目标地址的计算方法如下所示：

```
  0804 （源地址）
 +FFF6 （偏移量）
  07FA （目标地址）
```

注意：在把一个负偏移量加到源地址上时，就如上面所示一样，会在最高位上生成一个进位，该进位被丢弃。

3.2.6 绝对寻址

相对寻址方式仅在ACALL和AJMP两条指令中使用，都是2B指令，允许在当前2KB页面的程序存储器中调用或者转移，11位的目标地址由指令的低11位数据提供，也就是由操作码的A10~A8再加上指令第2字节的A7~A0构成（如图3-1f所示）。

指令中的高5位地址和当前程序计数器PC的高5位数值是相同的，所以分支指令的下一条指令及分支指令的目标地址都必须在同一2KB的页面内，因此A15~A11是固定不变的（如图3-3所示）。例如，如果标号THERE标志1条指令的地址是0F46H，且下面的指令

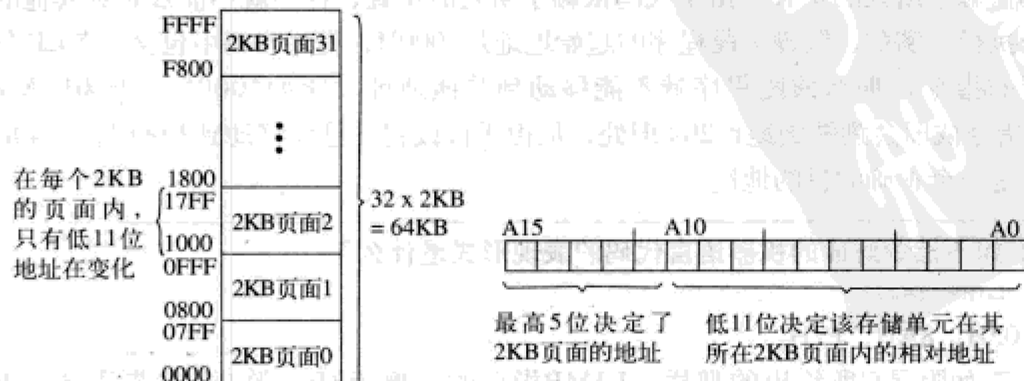
```
AJMP THERE
```

57

位于0900H和0901H，汇编器将该指令编译为

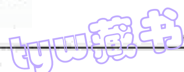
```
11100001 - 第1字节 (A10-A8+操作码)
01000110 - 第2字节 (A7-A0)
```

其中下划线的部分是目标地址的低11位，0F46H＝0000111101000110B。当执行该指令时，程序计数器的高5位是不变的。本例中的AJMP指令及其目标地址都位于0800H~0FFFH的2KB页面内，因此地址的高5位是公用的。



(a) 以2KB页面为单位显示的存储器图　(b) 在每个2KB页面内，源地址和目的地址的高5位是相同的，目的地址的低11位提供给指令使用

图3-3 绝对寻址方式的编码



绝对寻址可以通过简短的2B指令来实现程序的分支，但其寻址范围是受限的，而且其目标地址和程序代码在程序存储器中的位置相关。

例3-8　1条ACALL指令位于程序存储器的1024H和1025H，将要调用的子例程位于地址17A6H。该条ACALL指令的机器码是什么？

答案：F1H，A6H。

讨论：查阅附录C可知，ACALL指令的机器码为

```
aaa10001 aaaaaaaa
```

目标地址的低11位被间插在指令中，位10~8置于操作码的高3位，而位7~0被放置在指令的第2字节。本例中的目标地址表示成二进制格式如下所示，其低11位分成两组，其一是10~8的3个位，其二是7~0的8个位：

```
00010111 10100110 = 17A6H
.....aaa aaaaaaaa
```

接下来的任务就是按照规则将这11位地址安插到指令字节的正确位置，正确的排列顺序描述如下：

```
11110001 10100110 = F1A6H
aaa..... aaaaaaaa
```

注意：绝对寻址方式只能用在源地址和目的地址的高5位相同的情况下，其特征是源地址和目的地址必需落在同一个2KB页面内。

3.2.7　长寻址

长寻址方式只用于LCALL和LJMP两条指令中，是3B指令，其中的第2个字节和第3个字节合起来表示完整的16位地址（如图3-1g所示）。长寻址的优点在于，可以访问到全体64 KB程序存储器空间，缺点在于，指令的长度为3B，且采用长寻址的代码依赖于所处的位置。由于代码依赖于所处的位置，程序就不能放置到其他的地址处执行。例如，假设某段程序的起始地址是2000H，程序代码中包含一条LJMP 2040H的指令，那么该段程序就不能移动到其他地址，比如4000H。因为LJMP 2040H指令依旧会跳转到地址2040H处，但由于该段程序已经移动到4000H，2040H显然不是一个正确的目的地址。

58

例3-9　如下指令对应的机器语言代码的表现形式是什么？

```
LJMP 8AF2H
```

答案：02H，8AH，F2H。

讨论：正如附录C所给出的那样，LJMP指令的长度为3B，第1个字节是操作码(02H)，第2个字节和第3个字节联合起来，构成一个16位的目标地址，指令的第2个字节内是目的地址的高8位(8AH)，第3个字节内是目标地址的低8位(F2H)。

3.2.8 索引寻址

索引寻址把基址寄存器（程序计数器或者数据指针）中的数据和偏移量（位于累加器中）相加，所得到的和作为JMP或MOVC指令的有效地址（请参考图3-1h）。应用索引寻址，可以很容易地构造出跳转表和查找表。附录C中给出了MOVC A, @A+<base_reg> 指令和JMP@A+ DPTR指令的具体应用实例。

例3-10 如下指令的操作码是什么？

```
MOVC    A, @A+DPTR
```

答案： 93H。

讨论： 只需在附录C中查找MOVC指令，就能找到上面指令所对应的操作码。MOVC指令的长度只有1B，操作码不但规定了所要完成的功能，还规定了相应的寻址方式。该指令把位于程序存储空间中某个地址处的一个字节的数据传送到累加器中，把索引值（累加器的当前数据)和基址寄存器（数据指针）的值加起来所得到的和就是该地址。在执行完该指令后，索引值不再有效，因为已经被来自程序存储空间的数据所覆盖。

3.3 指令类型

59

8051的所有指令可以归纳为5种类型：

- ❑ 算术运算指令
- ❑ 逻辑运算指令
- ❑ 数据传送指令
- ❑ 布尔操作指令
- ❑ 程序分支转移指令

附录A按照功能分类给出了8051的指令速查表。读者在熟悉了8051的指令集后，会发现这张表很方便，可用来快速获取指令的参考信息。下面，本书将逐一探讨附录A中的各个类型的指令。

3.3.1 算术运算指令

在附录A中，所有的算术指令被分为一个组。由于ADD指令允许4种寻址方式，所以ADD A 指令有多种不同的形式：

```
ADD     A, 7FH     (直接寻址)
ADD     A, @R0     (间接寻址)
ADD     A, R7      (寄存器寻址)
ADD     A, #35H    (立即寻址)
```

几乎所有的算术指令都需要1个机器周期，但INC DPTR指令（需要2个机器周期）、MUL AB指令和DIV AB指令（需要4个机器周期）是例外。（注意，假如8051工作时钟的频率是12MHz，那么，1个机器周期的时间为1μs。）

例3-11 累加器中的数据为63H，寄存器*R*3的值为23H，当前的程序状态字寄存器PSW的值为00H。(a) 在执行完如下指令后，累加器和程序状态字PSW中的数据为多少？（以十六进制形式表示。）

```
ADD    A, R3
```

(b) 在执行完指令后，累加器中的数据为多少？（以十进制形式表示。）

答案：(a) A＝86H，PSW＝05H。(b) 以十进制表示的ACC＝？（请参考讨论）。

讨论：上面的例子看起来很简单：给定两个数，求二者的和。但其中包含了若干很有趣的概念，这些概念非常重要，读者务必要掌握。首先，把ACC和*R*3中的初始数据以十进制的形式表达出来，采用常规转换方式就可得到，A＝63H＝01100011B＝99_{10}，R3＝23H＝00100011B＝35_{10}，因而，加法结果应该是$99_{10}+35_{10}=134_{10}$。但是，问题来了。如果假设这是两个二进制补码形式的有符号数相加，那么，8位的数据，最大的正整数只能是127_{10}。如果是无符号数相加，8位的数据，最大的正整数是255_{10}，在这种情况下，上面计算得到的134_{10}正是所求的答案。

数据可能是有符号的二进制数、无符号的二进制数、二进制编码的十进制数或者ASCII码等。但是很显然，8051的CPU并不知道这些数据的区别，只有编程者确切地知道数据的差别所在。8051利用程序状态字寄存器PSW的“状态位”来管理不同类型的数据。具体情况请参看如下的二进制加法例子：

60

```
  11...11.
  01100011          (ACC=63H)
+ 00100011          (R3=23H)
  10000110      (ACC中的结果为86H )
```

结果是10000110B＝86H。请注意，执行加法运算时，在第0、1、5、6位上产生了进位，但在第2、3、4、7位上没有产生进位。由于在第7位上没有产生进位，因而，PSW的C（进位）标志在加法运算完成后没有被置位。

附录C中，关于加法指令对OV标志的影响有如下论述：“如果第6位有进位生成而第7位没有，或第7位有进位生成而第6位没有，OV将被置位，否则OV 被清零。”换成正规的说法就是：“对8位有符号数，如果结果越界，那么OV标志被置位。”在本例中，第6位有进位生成而第7位没有，因而，OV标志被置位。在处理有符号数的时候，OV标志被置位非常重要。因为此时数据允许的范围是$-128_{10}\sim+127_{10}$，而$99_{10}+35_{10}$的结果“越界了”。

从8051 CPU的角度来看，所处理的数据也可能是二进制编码的十进制数（只有编程者确切地知道数据是什么类型），因而，CPU也会相应地置1或者清除辅助进位

标志。本例中，在第3位上没有进位生成，因此，AC标志被清零。

最后还有一点，PSW中的P标志会根据累加器A中的数据的情况，置位或者清零，以建立起针对累加器的偶校验。由于加法运算的和保存在累加器中，累加器中有3个位是1，因此，P被置位，使得1的总个数为4，是偶数。PSW最终值为00000101B＝05H，只有OV标志和P标志被置位，其他位都被清零。

第二个问题是："在执行完指令后，累加器中的数据为多少？（以十进制形式表示）"，这个问题牵扯到一个很重要的概念：如何阐释像8051这样的CPU处理的数据？因为原来的问题没有提及数据的格式，或者说，不知道原来的数据该如何理解，所以，该问题没有确切答案，只能以问号来回答。话又说回来，至少存在两个可能的答案。首先，如果原来的数据是无符号二进制数，那么，运算的和"没有越界"（因为C＝0），答案是134_{10}。其次，如果原来的数据是有符号二进制数（采用补码的二进制数），那么，运算的和"越界"（因为OV＝1）。关键在于，与其说数据的具体含义（表达方法）不是CPU的特性，还不如说它是由软件管理数据的方式所决定的。

例3-12 写一段指令，实现从*R7*中减去*R6*，并把结果置于*R7*中。

答案：

```
MOV   A, R7
CLR   C
SUBB  A, R6
MOV   R7, A
```

讨论：无论是加法还是减法，总要有一个操作数预先置于累加器中。因而，上面的第1条指令负责把一个操作数传送到累加器中。第2条指令把程序状态字PSW中的进位标志清零。这一步是必须的，因为减法指令只有一种形式：带借位减法SUBB，该指令从累加器中减去源操作数和进位标志。对于减法来说，进位标志表示借位。如果在进行减法运算之前，进位标志的状态未知，那么必须用CLR C指令明确地把进位标志清零。第3条指令执行减法运算，并把结果存放于累加器中。第4条指令把减法运算的结果传送到寄存器*R7*。

61

8051提供了强大的寻址方式，用以访问其内部存储器空间。采用直接寻址方式，无需累加器作为中介，就可以对内部存储器空间任何地址处的数据进行递增或者递减操作。例如，如果内部RAM的7FH单元包含的数据为40H，那么指令：

```
INC 7FH
```

会把7FH单元的值增加到41H。

例3-13 假如8051中没有指令可以直接对内部RAM的数据进行递增操作。那么，如何实现相关的运算？

答案：

```
MOV   A, direct
INC   A
MOV   direct, A
```

讨论： 第1条指令把内部RAM某地址中的数据传送到累加器中。第2条指令将该数据（该数据现在位于累加器中）加1，第3条指令把递增结果写回到内部RAM中去。采用这种方式，比起完成同样功能的单条指令（INC direct）来说，指令序列较长，执行速度慢，而且会丢失累加器中原来的数据。

在INC类型的指令中，有1条指令用来处理16位的数据指针DPTR。由于数据指针中的数据是外部存储器空间的16位地址，因而，能在一步操作内对其进行递增操作的指令用处很大。但是，8051没有提供可以直接对16位数据指针进行递减操作的指令，要完成相应的功能，需要如下的指令序列：

```
        DEC    DPL              ;DECREMENT LOW-BYTE OF DPTR
        MOV    R7,DPL           ;MOVE TO R7
        CJNE   R7,#0FFH,SKIP    ;IF UNDERFLOW TO FF
        DEC    DPH              ;DECREMENT HIGH-BYTE TOO
SKIP: (continue)
```

DPTR的高位字节和低位字节必须分开来进行递减操作，但只有在低位字节（DPL）发生下溢（00H~FFH）的情况下，才对高位字节（DPH）进行递减操作。

MUL AB指令把累加器*A*中的数据和寄存器*B*中的数据相乘，16位的乘积保存在级联起来的寄存器*B*（乘积的高位字节）和累加器*A*（乘积的低位字节）中。DIV AB指令把累加器*A*中的数据除以寄存器*B*中的数据，把8位的商置于累加器*A*中，把8位的余数置于寄存器*B*中。例如，假设累加器*A*中数据为25（19H），寄存器*B*中的数据为6（06H），如下指令：

62

```
DIV   AB
```

把累加器*A*中的数据除以寄存器*B*中的数据，指令执行完后，累加器*A*中的数据为4，寄存器*B*中的数据为1 (25/6＝4余1) 。

例3-14　累加器中的数据为55H，寄存器*B*中的数据为22H，程序状态字PSW的值为00H。在执行完如下指令后，这3个寄存器的值变为多少？

```
MUL AB
```

答案： ACC＝4AH，*B*＝0BH，PSW＝05H。

讨论： 读者可以用计算器计算出乘法的结果，此外，把原来的数据以十进制的形式表示，ACC＝55H＝85_{10}，*B*＝22H＝34_{10}，那么乘积为$85_{10}\times 34_{10}=2890_{10}$＝0B4AH。高位字节（0BH）被置于寄存器*B*中，低位字节（4A）被置于累加器*A*中。PSW中的*P*标志被置位，同累加器一起构成偶校验。因为结果大于255_{10}，溢出标志被置位，因而PSW的值为05H。

例3-15 累加器A所包含的数据为1FH。寄存器B中的值最大能为多少才能使得在执行如下指令后，溢出标志OV不会被置位？

```
MUL  AB
```

答案： 08H。

讨论： 累加器A所包含的数据为1FH＝31_{10}，如果乘法运算的结果大于255，溢出标志OV就会被置位，和31_{10}相乘又使得乘积不超过255_{10}的最大整数为8_{10}＝08H（请注意，$31_{10}\times 8_{10}=248_{10}$，$31_{10}\times 9_{10}=279_{10}$）。

对于BCD（二进制编码的十进制数）数据的相关运算，在ADD指令和ADDC指令后，必须紧跟一条DA A指令（十进制调整指令），确保运算结果不越界。请注意，DA A不会把1个二进制数转换为BCD数，DA A指令仅能用在2个BCD码的加法运算之后做必要的调整。例如，假设累加器A中包含BCD数为59（59H），那么，如下指令序列：

```
ADD    A,#1
DA     A
```

先把1加到累加器A上，得到结果5AH，然后对A进行调整，得到的正确BCD结果是60（60H）（59＋1＝60）。

63

例3-16 写一段指令，演示如何把两个4位的BCD数加起来。假设第一个数位于内存空间的40H和41H处，第二个数位于内存空间的42H和43H处。两个数的最高位分别位于内存空间的40H和42H中，要求把加法结果存放到40H和41H中。

答案：

```
MOV     A,43H
ADD     A,41H
DA      A
MOV     41H,A
MOV     A,42H
ADDC    A,40H
DA      A
MOV     40H,A
```

讨论： 这是一个“多精度运算”的例子，8051的CPU被用来执行算术运算，而涉及的数据的位数大于CPU本身的位数（8051是8位）。虽然8051的算术指令一次只能对字节数据进行处理，但8051仍然可对更大的数据（例如，16位和32位的数据）进行加法和减法运算。二进制编码的十进制数（BCD）也是同样的道理，要把两个4位的BCD数加起来，需要进行两字节的加法运算。

多精度运算中的关键在于如何把进位从一个字节传到另一个字节。在加法运算

中，进位很自然地在字节中的各个位之间传递，但在把进位从一个字节传递到另外一个字节时，进位（如果有）可临时保存在PSW的进位标志C中。在处理低位字节的时候，不可能有先前运算产生的进位，因而，在执行第一个加法运算（处理低位字节）的时候，使用一般的ADD指令即可，但在执行第二个加法运算的时候，需要把低位字节可能产生的进位考虑进来，因此，必须使用ADDC指令。在处理二进制编码的十进制数（BCD）时，在两条加法指令之后，必须跟有DA A指令，DA A指令对加法运算结果做一些必要的调整工作。

3.3.2 逻辑运算指令

8051的逻辑运算指令（请参考附录A）对字节数据逐位进行布尔运算（“与”、“或”、“异或”、“非”等）。如果累加器中的数据为00110101B，那么，如下的“与”逻辑运算指令：

```
ANL   A,#01010011B
```

使累加器的值变成了00010001B。运算过程如下所示：

```
        01010011   (立即数)
AND     00110101   (累加器A中原来的数据)
        00010001   (累加器A中的逻辑运算结果)
```


对于逻辑运算指令来说，其寻址方式和算术运算类指令相同，因而，ANL指令有几种形式：

```
ANL   A,55H      (直接寻址)
ANL   A,@R0      (间接寻址)
ANL   A,R6       (寄存器寻址)
ANL   A,#33H     (立即寻址)
```

所有把累加器中的数据作为一个操作数的逻辑运算指令执行该指令都需要花费1个机器周期，其他的逻辑运算指令需要花费2个机器周期。

不需要累加器A作为中介就可以对内部数据存储空间中的任何数据进行逻辑运算。XRL direct, #data指令可以方便快速地把端口数据取反，例如：

```
XRL  P1,#0FFH
```

上面的指令执行时，进行读－改－写操作。首先，读取端口1中的8位数据，然后，将每位数据和立即数中的相应位进行“异或”运算，由于立即数的8位数据都是1，因而，此处的异或运算实际上就是求反（$A\oplus 1=\overline{A}$）。最后，将异或运算的结果写回到端口1中。

循环移位指令（RL A和RR A），把累加器中的所有数据位向左或向右移动一位。对于循环左移来说，第7位的数据被移到第0位中。对于循环右移来说，第0位的数据被移到第7位中。带进位的循环移位指令RLC A和RRC A，把累加器中的数据位

连同PSW中的进位标志（一共9位）向左或向右移动1位。例如，假设进位标志的值是1，累加器A中的数据为00H，那么，指令

```
RRC A
```

执行后，进位标志变为0，累加器A变成80H。进位标志的值被移动到ACC.7中，而ACC.0被移动到进位标志中。

SWAP A指令交换累加器中数据的高4位和低4位。这条指令在处理BCD数的时候很有用。例如，假设累加器的值小于100_{10}，下面的指令序列可以快速地把累加器中的数据转为BCD数：

```
MOV  B,#10
DIV  AB
SWAP A
ADD  A,B
```

前面2条指令把累加器中的数除以10，这样，累加器中数据的低4位是十位上的数字，寄存器B中数据的低4位是个位上的数字。SWAP 指令把十位上的数字移动到累加器的高4位，ADD指令把个位上的数字装载到累加器的低4位。

例3-17 写两段指令，用两种方式演示如何把累加器中的各位数据循环左移3位。并从存储空间需求和执行速度方面讨论这两种方法的优缺点。

答案：

```
(a) RL    A
    RL    A
    RL    A
(b) SWAP  A
    RR    A
```

讨论：（a）是最容易想到的方法，相比之下，方法（b）用了一个小技巧。如附录C中所示，SWAP A指令负责交换累加器中数据的高4位和低4位，这等价于把累加器中的数据循环左移4位，为了抵消第4次循环左移，需要1条循环右移指令（RR A）。

虽然方法（a）和方法（b）的效果一样，但在对内存的需求和执行速度上有着细微的区别。上面所有的指令都是1字节长度的指令，执行时间都是1个机器周期（请参考附录C）。但方法（a）会在存储空间中占据3个字节，且执行起来需要耗费CPU的3个机器周期。方法（b）仅需2个字节的存储空间，执行时间仅需要2个机器周期。两种方法的差别看上去很细微，但在嵌入式控制系统场合，设计时常常需精简程序，只要有改进，甚至是微小的改进都是必要的。

例3-18 写一段指令，颠倒累加器A中数据的各个位的次序。也就是说，第7位和第0位互换，第6位和第1位互换，等等。

答案：

```
            MOV    R7,#8
LOOP:       RLC    A
            XCH    A,0F0H
            RRC    A
            XCH    A,0F0H
            DJNZ   R7,LOOP
            XCH    A,0F0H
```

讨论：这个例子中用到的技巧有时被称为“比特舞”。上面的例子看起来有点像是故意设计的，但经常应用在要求灵巧地处理字节中各个位的位置的场合，而在这样的场合，其出现的频率高得令人震惊！

上面提供的解决方案就是在寄存器*B*中创建新值，途径是逐一把累加器中的数据移到进位标志中，然后，再把进位标志逐一移回寄存器*B*中。为了改变位的模式，第一次移位是“到左边”，第二次移位是“到右边”。由于循环移动指令仅能操作于累加器上，因而，在每次循环移动的时候，需要交换（XCH）累加器和寄存器B中的数据。最后一个XCH指令调整累加器中的正确结果。注意：寄存器*B*位于直接地址0F0H。

3.3.3 数据传送指令

1. 内部RAM

用来传递内部存储器空间中的数据的指令（请参考附录A），执行起来需要耗费1个或2个机器周期。这些指令的格式如下：

66

```
MOV <destination>, <source>
```

该类指令无需累加器作为中介，就可以把数据在内部RAM或特殊功能寄存器SFR中的任意两个单元之间进行传送。要牢记，8032/8052的数据RAM的高128B只能采用间接寻址来访问，而各个特殊功能寄存器SFR只能通过直接寻址来访问。

MCS-51的体系结构中，有一个不同于绝大多数微处理器的特性：栈位于片上RAM中，并且其增长方向是向上生长。PUSH指令先是递增栈指针SP的值，然后将数据复制到栈中。PUSH和POP指令采用直接寻址方式确认数据是被保存还是被恢复，但栈指针的数值只能通过SP寄存器来间接访问。这意味着在8032/8052上栈可能会占据到内部存储器的高128B。

内部存储器的高128B在8031/8051中不能被访问。在这类器件中，如果SP的值超过7FH(127)，那么，接下来的PUSH指令所要保存的数据都会丢失，而POP指令将会弹出不确定的数据。

例3-19　栈指针SP的值是07H，累加器*A*中的数据是55H，寄存器*B*中的数据是

4AH。那么，在执行完如下指令序列后，内部RAM中的什么地址单元将被改变？其新值是多少？

```
PUSH    ACC
PUSH    0F0H
```

答案：

Address	Contents
08H	55H
09H	4AH
81H(SP)	09H

讨论：第1条指令把累加器中的数据压入栈中。在把数据复制到栈中之前，8051先递增栈指针的值，因而，累加器*A*中的数据（55H）被写到内部RAM地址08H单元。第2条指令把寄存器*B*（位于内部RAM地址F0H处）中的数据压入到栈中，同样，在把数据复制到栈中之前，8051先递增栈指针的值，因而，寄存器*B*中的数据（4AH）被写到内部RAM地址09H处的字节中。两条PUSH指令递增两次栈指针的值，因此，SP的最终值是09H。

数据传送指令包含1条16位的MOV指令，用来初始化数据指针DPTR，如果要在程序存储空间中进行查找表操作，或者要访问16位的外部数据存储空间，都需要用到该指令。

例3-20　数据指针寄存器在内部RAM中的地址是多少？

答案：数据指针的高位字节DPH的地址是83H，低位字节DPL的地址是82H。

讨论：数据指针寄存器是一个16位的寄存器，在内部RAM中占据2个字节。高位字节DPH位于地址83H处，低位字节DPL位于地址82H处。

如下格式的指令：

```
XCH    A,<source>
```

交换累加器和由<source>所直接寻址的字节中的数据。类似的，8051中包括交换“数据”的指令，格式如下：

67

```
XCHD   A,@Ri
```

但上述指令只会把两个操作数的低四位进行交换。例如，如果累加器中的数据是F3H，寄存器*R*1中的数据是40H，在内部RAM中，40H单元的数据为5BH，那么，指令

```
XCHD   A,@R1
```

执行完毕后，*A*的内容变成了FBH，内部RAM 40H单元的内容变成了53H。

例3-21　在执行完如下指令序列后，累加器*A*和*B*中的数据各是多少？

```
MOV       0F0H,#12H
```

```
MOV     R0,#0F0H
MOV     A,#34H
XCH     A,0F0H
XCHD    A,@R0
```

答案：$A=14H$，$B=32H$。

讨论：在本例中，前3条指令用来设定各个累加器的值，$A=34H$，$B=12H$，$R0=F0H$。请注意，累加器B位于内部RAM地址F0H。第4条指令交换累加器A和B的内容，使得$A=12H$，$B=34H$。第5条指令交换累加器A和累加器B的低4位，使得$A=14H$，$B=32H$。

2. 外部RAM

在内部存储器空间和外部存储器空间传送数据的指令，采用间接寻址方式。访问的是1字节地址（@Ri，其中Ri是当前工作寄存器组的$R0$和$R1$）或者2字节地址（@DPTR）。采用16位地址（2字节）的不利之处在于，端口2的8个位被用于输出16位地址的高位字节，因此，端口2暂时不能被用作I/O口。但在另一方面，倘若采用8位地址，虽然没有占用端口2，但是仅能访问内部RAM的有限地址单元（参考第2章）。

所有访问外部存储空间的数据传送指令，执行时需要消耗2个机器周期，而且，累加器A或者被用作源操作数，或者被用作目的操作数。

读信号（$\overline{RD}$）和写信号（$\overline{WR}$）仅在8051读写外部RAM的时候（执行MOVX指令）才会被激活。正常情况下，这两个信号处于被禁用状态（高电平），如果设计中没有用到外部数据存储，这两个引脚可以用作I/O端口。

例3-22　写一段指令，演示如下功能，读取外部RAM地址10F4H和10F5H单元的数据，然后分别存放到寄存器$R6$和$R7$中。

68

答案：

```
MOV     DPTR,#10F4H
MOVX    A,@DPTR
MOV     R6,A
INC     DPTR
MOVX    A,@DPTR
MOV     R7,A
```

讨论：第1条指令把数据指针DPTR的值初始化为待读取的第1个外部RAM地址10F4H，第2条指令从外部存储空间地址10F4H单元读取第1个字节的数据，并把读取的数据存放在累加器中。第3条指令把读取到的数据传送到寄存器$R6$中。请注意，MOVX指令必须将累加器用作源操作数或者目的操作数。第4条指令递增数据指针DPTR的值，使其指向待读取的第2个地址10F5H，第5条指令从外部存储器空间地址10F5H单元读取1个字节的数据，并将其存放到累加器中。第6条指令把读取到的数据传送到寄存器$R7$中。

3. 查表

在8051中，有2条数据传送指令可用于在程序存储空间中执行查表操作。由于是访问程序存储空间，数据是只读的，所以不能对其进行更新操作。指令的助记符是MOVC（含义是“移动常数”）。MOVC采用索引寻址，累加器*A*中的数据是偏移量，基址寄存器是程序计数器或者数据指针。

如下指令：

```
MOVC    A,@A+DPTR
```

可以访问一个包含256项入口的数据表（入口编号从0~255）。在工作时，待访问入口的编号被装入累加器*A*中，然后初始化数据指针DPTR，使其指向数据表的起始地址。指令：

```
MOVC    A,@A+PC
```

有着同样的工作原理，但是以程序计数器PC的内容作为基址。通常情况下，通过调用子例程来访问数据表。首先，待访问入口的编号被装入累加器*A*中，然后调用子例程。相关的设置和调用指令序列如下：

```
MOV     A, #ENTRY_NUMBER
CALL    LOOK_UP
 .
 .
 .
LOOK_UP:  INC   A
    MOVC A,@A+PC
    RET
TABLE:    DB   data, data, data, data, ...
```

在上面程序中，数据表位于程序存储空间中RET指令之后。INC指令是必需的，因为在执行MOVC指令时，PC指向RET指令。INC递增累加器*A*的值，实际效果是在进行查表操作时，自然地越过RET指令，直接找到数据表的起始地址。

但要注意，即使我们希望通过使用这种技术得到一张有256个入口的表，但最终也只能得到255个入口，因为运行INC A指令会丢失一个入口。下面谈谈被移入累加器的这个数字——255。INC A指令加1，使得值255回零，因为累加器是8位的。下一条指令（MOVC A, @A＋PC）将试图把RET的值载入累加器，这是无效的。因此，有效的入口只是0~254。

69

例3-23 写一个SQUARE子例程，计算0~9整数的平方值。要求调用子例程前，把待求的整数装入到累加器*A*中，子例程返回时，该整数的平方值置于累加器*A*中。写出以下子例程：（a）使用查表法；（b）不使用查表法；（c）调用SQUARE子例程，计算整数6的平方为36。

答案：

（a）使用查表法的程序

```
SQUARE: INC     A
        MOVC    A, @A+PC
```

```
        RET
TABLE:  DB    0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

(b) 不使用查表法的程序

```
SQUARE: PUSH  0F0H
        MOV   0F0H, A
        MUL   AB
        POP   0F0H
        RET
```

(c) 调用子例程

```
        MOV   A,#6
        CALL  SQUARE
```

讨论：若要在程序存储器空间中实现查表，(a) 中的子例程和 (c) 中的调用指令是一种比较直观的解决方案。但在例3-23中，还存在另外一种颇有趣的解决方案，如 (b) 中程序所示，把累加器A的数据复制到寄存器B中（位于内部RAM地址F0H），然后，MUL AB指令把累加器A的值和寄存器B的值相乘，乘积被置于累加器A中，该数据就是所要求的平方值。由于寄存器B的内容会被第2条指令所改写，因而，需要先把寄存器B的值保存在栈上，在从子例程返回之前，从栈上弹出前面保存的数据到寄存器B中。PUSH/POP 指令不一定是必需的，使用与否取决于上下文，但必须指出，在设计子例程的时候，要将副作用尽可能地降低，这是一种非常好的程序设计习惯。

比较方案 (a) 和方案 (b)，会发现这两个方案的优劣性需要仔细权衡。方案 (a) 中，指令加上数据表，共占用了13B，而方案 (b) 只占用了8B。但是，方案 (a) 中的指令执行1次只要5个机器周期，而方案 (b) 中的指令执行1次需要11个机器周期。方案 (a) 的执行速度快（这是其优点），但要消耗较多的存储空间（这是其缺点），相比之下，方案 (b) 执行速度慢（这是其缺点），但是对存储空间的要求比较小（这是其优点）。如果查找表比较大，每1项都会占用1B，那么，如何在这两种方案之间做出取舍就更难了。

在很多情况下，查找表的索引值和表中各元素之间的关系没有这个例子所展示的那样简单，而且查找表往往是唯一可行的实现方案。

70

3.3.4 布尔操作指令

为了进行单个位的运算，8051处理器配有一个功能齐全的布尔处理器。内部RAM包括128个可寻址的位，SFR空间还另外支持数量高达128的可寻址位。所有的端口线都是可位寻址的，每条端口线可以作为一个单独的“位”进行处理。访问这些“位”的指令不只有分支类指令，还有个完整的指令集合，功能包含传送、置位、清零、取反、或运算和与运算等。MCS-51™系列微控制器最强大的特性之一，就是可以执行“位”的操作，对于那些面向字节操作的微处理器来说，其架构决定了要做到此点很不容易。

附录A中列出了8051现有的布尔操作指令。所有访问“位”的操作均采用直接寻址方式，其中内部RAM有128个可直接寻址的位，位地址是00H~7FH，而SFR空间中的128位，其位地址是80H~FFH。低端的128位，其对应的字节地址是20H~2FH，各个“位”按序排列，字节20H的第0位的位地址是00H，字节2FH的第7位的位地址是7FH。

单条指令就可以设置或清除位。8051常常需要对单个“位”进行读写，以便控制许多I/O器件，在输出方面，包括中继器、电动马达、螺线管、LED状态灯、蜂鸣器、警报器、扬声器等；而在输入方面，包括多种类型的开关和状态指示器。例如，假设有一个警报器被接到端口1的第7位上，那么，如下设置该端口位，会打开该警报器：

```
SETB P1.7
```

清除端口位，则关闭该警报器：

```
CLR P1.7
```

汇编器会负责把符号P1.7转换成正确的位地址97H。

注意以下指令是如何轻松地把8051内部的标志转移到端口引脚上的：

```
MOV  C, FLAG
MOV  P1.0, C
```

在本例中，FLAG是低端128位或SFR空间中的任意一个可寻址的位，根据FLAG 对应的位是1还是0，将某条I/O信号线（本例中是端口1的第0引脚）置1或者清零。

在8051的布尔处理器中，程序状态字PSW的进位标志C被用作位运算中的“累加器”。在位操作指令中，如果涉及进位标志C，那么该指令在汇编的时候，等同于那些进位相关的指令（例如CLR C）。PSW寄存器是一个可位寻址的寄存器，因而，进位标志C也有位地址。就像其他可位寻址的SFR一样，PSW中的各个位都有预定义的助记符，汇编器在位操作的场合，能够接受这些助记符作为输入。进位标志的助记符为CY，位地址为0D7H。下面两条指令：

```
CLR        C
CLR        CY
```

的执行效果是一样的，但前者是1B的指令，而后者是2B的指令。在后者的机器代码中，第2个字节是进位标志位CY的位地址。

对于单个“位”变量来说，诸如AND、OR、NAND、NOR和NOT这样的逻辑运算，实现起来很容易，而且，位变量可以是8051 I/O端口的输入或输出信号线。因而，8051可以直接读取I/O引脚上的电平状态，而布尔操作指令可以直接改变I/O引脚上的电平状态。假如要对端口1的第1位和第0位进行逻辑AND运算，并把结果输出到端口1的第2位上（逻辑关系如图3-4所示），那么，相应的指令序列如下：

```
LOOP:   MOV  C, P1.0        ;n1 = 1 cycle
        ANL  C, P1.1        ;n2 = 2 cycles
        MOV  P1.2, C        ;n3 = 2 cycles
        SJMP LOOP           ;n4 = 2 cycles
```

8051连续不断地读取P1.1和P1.0的状态，进行逻辑AND运算后，又输出到P1.2上。

如果图3-4中的逻辑运算采用典型的电子逻辑线路来实现，比如说采用74AL508，那么从输入到输出的传播延迟是7ns。传播延迟是指从某个输入信号发生变化、到输出端口处建立起正确的电平所需要经历的时间。如果图3-4中的逻辑关系采用上面的软件来实现，那么，最坏情况下的传播延迟是多少呢？请参考图3-5，该图有助于回答这个问题。最坏的情况是，P1.0在第1条指令刚执行完时发生变化（如图3-5中的A所示），该变化要被察觉到，需要等到下一轮循环开始后，而输出端口上出现正确电平的时刻是图3-5中的B。图3-5中的实线标出了在最坏情况下所需经历的指令序列。从A到B，CPU需要花费11个机器周期的时间，在12MHz的时钟下，时间延迟最大可达11μs。很明显，微控制器的速度根本不能和电子逻辑线路的速度相比，用74ALS08与门来实现图3-4中的逻辑运算比软件快1000倍。

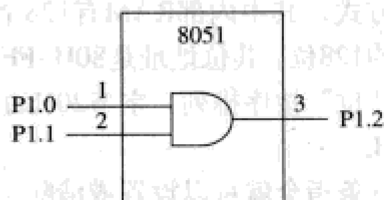


图3-4 与门逻辑关系的简单实现

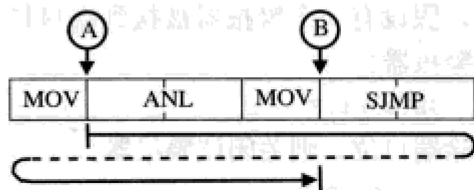


图3-5 传播延迟最坏情况示意图

注意，位的布尔运算指令包括ANL（与逻辑）和ORL（或逻辑），不包括XRL（异或逻辑），但异或逻辑很容易实现。例如，假设要求对两个位BIT1和BIT0进行异或运算，并把结果置于进位标志中，实现的指令如下：

72

```
          MOV    C,BIT1
          JNB    BIT2,SKIP
          CPL    C
SKIP:     (continue)
```

首先，把BIT1传送到进位标志中，如果BIT2＝0，那么进位标志C就是所求的结果，因为在BIT2＝0时，BIT1⊕BIT2＝BIT1。如果BIT2＝1，那么，进位标志C是正确答案的补，对C求补，就得到了正确答案。

位测试

上例的程序中用到了JNB指令，该指令属于位测试指令系列。如果被测试的位是1，这些指令的一部分就跳转（JC，JB，JBC）；如果被测试的位是0，其余指令就跳转（JNC，JNB）。在上例中，如果BIT2＝0，那么CPU会跳过CPL指令。JBC指令在被测试位为1的情况下跳转，但在跳转的同时，会把测试位清零。因而，在单条指令中，可以同时测试且清零某个标志位。

PSW中的各个位是可直接寻址的，如校验位和通用标志位都支持位测试指令。

3.3.5 程序分支转移指令

如附录A所示，8051包括大量可控制程序流程的指令，包括调用子例程和从子例程返回的指令、条件转移指令以及无条件转移指令。但情况更复杂的是，程序分支转移指令用到了3种寻址模式。

JMP指令有3种变体：SJMP、LJMP和AJMP(分别使用相对寻址、长寻址和绝对寻址)。如果程序编写者对具体使用哪种变体不介意，Intel的汇编器(ASM51)允许在源代码中使用泛型的JMP助记符。但其他公司的汇编器不一定支持这个功能。对于泛型的JMP指令来说，在汇编的时候，如果目标地址不是前向引用，且目标地址和紧接JMP的下一条指令的地址位于同一个2KB页面内，那么JMP指令被汇编为AJMP指令。否则，JMP指令被汇编为LJMP 指令。泛型的CALL指令的工作原理是一样的。

就如前面寻址模式一节中讨论的那样，在SJMP指令中用相对偏移量来标识目标地址。由于SJMP指令是2B长度的指令（1B的操作码加上1B的相对偏移量），因而，对于跳转的目标地址来说，其范围限制在紧接着SJMP的下一条指令地址−128B~+127B的范围内。

LJMP指令用16位的常数来标识目标地址，由于LJMP指令是3B长度的指令（1个字节的操作码加上2个用于存放地址的字节），因而，目标地址可以位于64KB程序存储器空间中的任何一处。

73

AJMP指令用11位的常数来标识目标地址，同SJMP一样，AJMP也是2B长度的指令，但指令的二进制编码方式不同。指令的第1个字节（操作码字节）包含11位地址的高3位，指令的第2个字节包含11位目标地址的低8位。在执行AJMP指令的时候，8051用这11位数据来代替程序计数器PC的低11位，但PC的高5位不变。因此，目标地址必须与紧接AJMP指令的下一条指令的地址处于同一2KB页面内。由于代码存储器空间是64KB，因而，总计有32个这样的页面，每一内存页的起始地址位于2KB地址的边界上（0000H，0800H，1000H，1800H……一直到F800H，请参考图3-3）。

在上面的3种情况下，编程者都是依照通常方式来定义目标地址的，或者采用标号，或者采用16位的常数。汇编器会把目标地址转换为能够插入到该跳转指令的正确格式。如果目标地址太远，超出了指令所要求的格式能够支持的范围，那么汇编器会给出一个“目标地址越界”的提示。

1. 跳转表

指令JMP @A+DPTR支持跳转表的无条件跳转。其目的地址由当前时刻的DPTR寄存器和累加器的数值的和确定。一般情况下，DPTR中存放的是表格的首地址，累加器中放的是索引地址。例如，有5个待转移的标号地址，将0~4这5个数值依次放入累加器，执行下面的指令即可完成到适当位置的跳转：

```
MOV    DPTR, #JUMP_TABLE
MOV    A, #INDEX_NUMBER
RL     A
JMP    @A+DPTR
```

其中的RL A指令将索引值（0~4）转换成0~8之间的偶数，这是因为每条AJMP指令的地址都是2B的：

```
JUMP_TABLE:    AJMP   CASE0
               AJMP   CASE1
               AJMP   CASE2
               AJMP   CASE3
```

例3-24　假设上面的跳转表在程序存储器中的首地址是8100H，具体存储细节如下：

地址	内容
8100	01
8101	B8
8102	01
8103	43
8104	41
8105	76
8106	E1
8107	F0

(a) 该段指令所在的2KB程序存储器页面的起始地址和终止地址是什么？

(b) 程序中CASE0到CASE3在什么地址开始？

答案：

(a) 8000H~87FFH。

(b) CASE0在80B8H地址开始。

CASE1在8043H地址开始。

CASE2在8276H地址开始。

CASE3在87F0H地址开始。

74

讨论：这个例子实际上更多是练习绝对跳转指令的。跳转表包括4条AJMP指令。由于是采用绝对寻址，那么跳转指令的目标地址应当和紧接AJMP的下一条指令的地址位于同一个2KB页面内。在存储空间的1个2KB页面中，地址的高5位一样的。8100H的高5位是10000B，而每一个CASE项里的AJMP指令中包含了AJMP指令的低11位。其中，有3位包含在AJMP指令的操作码字节中，剩下的8位在AJMP指令的第2个字节中。例如AJMP CASE3指令，该指令的2个字节在存储空间中的地址是8106H（E1H）和8107H（F0H），第1个字节（操作码字节）的高3位是111B，第2个字节是11110000B，把这3段二进制码按照规则连接起来，就得到了CASE3的目标地址10000 111 11110000B＝87F0H。

2. 子例程和中断

CALL指令有两个变体：ACALL和LCALL，分别使用绝对寻址和长寻址。同JMP指令一样，如果程序编写者对目标地址如何编码不介意，Intel的汇编器（ASM51）允许在源代码中使用泛型的CALL助记符。两条指令在执行的时候，都会把当前程序计数器PC的值压入栈中，然后把子例程的地址装入程序计数器中。注意，压入栈的PC值，是紧接CALL指令的下一条指令的地址。在压栈的时候，先是压入PC的低位字节，再压入PC的高位字节。从栈中弹出的顺序正好与此相反。例如，假设某条LCALL指令在代码空间中的地址是1000H~1002H，而当前的SP的值是20H，那么执行该LCALL指令：(a) 把返回地址1003H压入栈中，内部RAM 21H单元的数据是03H，22H单元的数据是10H；（b）使SP的值保持在22H不变；(c) 把位于LCALL指令的第2个字节和第3个字节中的子例程地址装入程序计数器PC中，并跳转到该地址去执行子例程。

例3-25　如下指令：

```
LCALL COSINE
```

位于代码空间中的地址0204H~0206H单元，而子例程COSINE在代码空间中的起始地址是043AH。假设在执行上面的指令之前，栈指针SP的值为3AH，那么执行LCALL指令后，内部RAM中何处的数据被修改了？修改后新数值是多少？

答案：

地址	内容
3BH	07H
3CH	02H
81H (SP)	3CH

讨论： LCALL指令的长度是3个字节，紧接在LCALL指令后的下一条指令的地址是0207H，在执行完子例程COSINE后，8051必须回到该地址继续执行原来的程序，8051在分支转移去执行子例程之前，必须把该地址保存到栈中。由于栈指针SP的初值为3AH，且8051会先把栈指针的值增加1，然后再把数据保存到SP指针所指向的数据单元中，因而，返回地址（0207H）的高位字节02H被保存到内部RAM的3BH单元，而其低位字节07H被保存到3CH单元。在执行LCALL指令后，SP的值被增加到3CH。注意，SP是一个特殊功能寄存器，其在内部RAM中的地址为81H。

在LCALL指令和ACALL指令中，目标地址所受到的限制和前面刚刚讨论过的LJMP指令和AJMP指令是一样的。

执行完子例程则要使用RET指令，确保8051能够返回到CALL指令的下一条指令的地址去继续执行程序。RET指令如何使得8051返回到主程序中呢？把栈最顶端

的两个数据弹出到程序计数器PC中即可。在程序设计的时候，若是要用到子例程，设计者必须牢记一条原则：在任何情况下，都要用CALL指令进入子例程，用RET指令退出子例程。采用其他方式进出子例程，通常会破坏掉栈中的数据，致使程序崩溃。

RETI用来退出中断服务程序（ISR）。RET指令和RETI指令的唯一区别在于，RETI指令会通知中断控制系统，当前中断已经处理完毕。如果此刻没有其他中断在等候处理，那么RETI的动作和RET的一样。有关中断和RETI指令的更详细的讨论见第6章。

例3-26　在执行如下指令前，栈指针SP的值是1FH，在执行完如下指令后，SP的值变成多少？

```
RET
```

答案： 1CH。

讨论： 子例程的返回地址是16位（2字节）的。RET指令从栈的顶部取出两字节的数据，并把数据置于程序计数器中，使8051能够从CALL指令随后的指令地址处继续执行程序。不管返回地址是多少，与RET指令执行前的SP值相比，栈指针SP的值减少了2。

例3-27　位异或

8051指令集没有进行两个位数据异或逻辑运算的指令。编写名为XRB的子例程，以XRB C，P的格式进行两个位数据的异或操作。两个待进行异或操作的位数据分别预先存放在C和P中，该子例程被调用之后，计算结果存放到C中。

76

答案：

```
XRB:        MOV  20H, C        ;backup the first bit, x
            ANL  C, /P         ;C = x̄y
            MOV  21H, C        ;backup partial result, x̄y
            MOV  C, P          ;put second bit, y in C
            ANL  C, /20H       ;C = ȳx
            ORL  C, 21H        ;C = ȳx + x̄y
```

讨论： 本程序通过3个步骤解决了问题，其依据就是异或运算的关系式 $x \oplus y = \bar{x}y + \bar{y}x$。其他两种解决方案在课后习题中将涉及，读者可自行练习：

- ❑ 将位数据插入到字节数据中，然后完成两个字节的异或运算；
- ❑ 使用JB和JNB指令。

3. 条件跳转

8051提供了多种类型的跳转指令。所有条件跳转指令的寻址方式都是相对寻址，因此，以紧接条件跳转指令的下一条指令的地址为基准，跳转的范围限制在

−128B~+127B之内。但有一点要注意：程序编写者定义目的地址的方式和其他跳转指令是一样的，或者用标号，或者用常数，剩下的事情由汇编器来处理。

在程序状态字PSW中没有0标志位。JZ和JNZ指令会测试累加器A中的数据，看其是否为0。

DJNZ指令（递减，若非0跳转）用在循环控制中。如果要执行某个循环N次，把某个用于计数的数据单元置为N，在循环的结束处，用DJNZ指令来判断是否要重新开始循环，指令如下（$N=10$）：

```
         MOV   R7, #10
LOOP:    (begin loop)
         .
         .
         .
         (end loop)
         DJNZ R7, LOOP
         (continue)
```

CJNE指令（比较，若不等则跳转）也可用在循环控制中，该指令有两个1字节的操作数，仅在这两个操作数不相等的情况下，程序才会发生跳转。例如，假设从串行端口读一个字符到累加器A中，如果该字符是CONTROL-C（03H），那么CPU会跳转到标号TERMINATE处去执行程序。指令序列如下：

```
          CJNE A, #03H, SKIP
          SJMP TERMINATE
SKIP:     (continue)
```

77

因为仅在A≠CONTROL-C的情况下，程序才会跳转到标号SKIP处去，所以程序在字符不相等的情况下，越过SJMP TERMINATE指令继续往下执行。

CJNE指令也可以用在“大于”和“小于”的比较场合。两个操作数被当作无符号数，如果第1个数小于第2个数，那么进位标志C被置1，如果第1个数大于第2个数，那么进位标志C被清零。例如，程序要求累加器A中的值大于或等于20H，CPU就跳转到BIG处去执行指令，那么，相关指令如下：

```
CJNE    A, #20H, $+3
JNC     BIG
```

CJNE跳转指令的目标地址是“$+3”，$是一个特殊符号，代表当前指令的地址。CJNE是一个3字节长度的指令，因此，“$+3”代表下一条指令JNC的地址。换句话说，不管CJNE指令的比较结果如何，CPU接下去总是会执行JNC指令。CJNE指令比较两个操作数唯一的作用是把进位标志C置1或清零。JNC指令会根据进位标志C的状态决定程序跳转与否。本例表明了一点，比起大部分微处理器来说，8051实现某些程序设计功能比较笨拙。但是，读者在第7章中会看到，应用8051汇编语言中的宏，可以把功能强大的指令序列（就如上面所示的例子一样）简写成一个助记符，8051汇编语言允许该助记符用于程序设计之中。

小结

本章介绍了8051的指令集。推荐读者参阅书后的附录C来学习更多的相关指令的例程。当然，由于缺乏实践锻炼，读者应该通过尽可能多地钻研例程来达到掌握指令的目的。随后的3章中将讨论更多的关于8051片上资源的例程，包括定时器/计数器、串行端口和中断。

习题

3.1 如下指令所对应的机器码的十六进制表现形式是什么？

(a) INC DPTR

(b). MOV A,#-2

(c) MOVX @DPTR,A

(d) CJNE A,#0DH,$+3

(e) PUSH ACC

(f) SETB P2.2

3.2 如下指令所对应的机器码的十六进制表现形式是什么？

(a) MOV DPH,#84H

(b) JNB ACC.0,$

(c) POP DPH

(d) MOV A,# '='

(e) XLR A,#'S'

(f) CLR C

3.3 下面的机器语言代码，各对应什么汇编指令？

(a) 7EH, 02H

(b) C2H, 97H

(c) 13H

(d) F6H

(e) 22H

(f) 90H, 80H, 30H

3.4 下面的机器语言代码，各对应什么汇编指令？

(a) EFH

(b) 12H, 80H, 50H

(c) F5H, 8DH

(d) 04H

(e) 83H

(f) 75H, 8AH, E7H

3.5 列出8051中所有长度为3字节、且对应的机器语言代码的操作码以5H结尾的指令。

3.6 列出8031中所有长度为2字节、且对应的机器语言代码的操作码以2H开头的指令。

3.7 写一段指令，采用间接寻址方式，演示如何把内部RAM空间地址50H处的数据传送到累加器A中。

3.8 用两种方式来说明，如何把累加器A中的数据传送到内部RAM的地址50H单元。

3.9 在8051中，什么样的操作码没有得到定义？

3.10 在8052中，定义了多少种操作码？

3.11 8051指令如下所示：

```
MOV    50H,#0FFH
```

(a) 该指令对应的操作码是多少？

(b) 这条指令是几字节长度的指令？

(c) 说明指令中的各个字节起什么作用。

(d) 执行该指令需要耗费多少个机器周期？

(e) 如果8051的晶振是16MHz，那么，执行该条指令需要耗费多长时间？

3.12 8051指令如下所示：

```
CJNE    A,# 'Q',AHEAD
```

(a) 该指令对应的操作码是多少？

(b) 这条指令是几字节长度的指令？

(c) 说明指令中的各个字节起什么作用。

(d) 执行该指令需要耗费多少个机器周期？

(e) 如果8051的晶振是16MHz，那么，执行该条指令需要耗费多长时间？

3.13 指令

```
SJMP AHEAD
```

的相对偏移量是多少？假设该指令位于代码空间中地址 0400H和0401H处，标号AHEAD代表位于地址041FH处的指令。

3.14 指令

```
SJMP BACK
```

的相对偏移量是多少?假设该指令位于代码空间中地址A050H和A051H处，标号BACK代表位于地址9FE0H 处的指令。

3.15 假设指令

```
AJMP AHEAD
```

位于代码空间中地址2FF0H和2FF1H处，标号AHEAD代表位于地址2F96H处的指令，那么，上面指令所对应的机器语言代码的十六进制表示形式是什么？

3.16 假设指令

```
ACALL FACTORIAL
```

位于代码空间中地址06F4H和06F5H处，符号FACTORIAL代表一个子例程，该子例程在代码空间中的起始地址是07ABH，那么，上面指令所对应的机器语言代码的十六进制表示形式是什么？

3.17 假设在程序中的某一点，如果累加器A中的值等于回车符号的ASCII码，那么8051就要跳转到标号EXIT处去。要实现如上功能，相应的指令序列是什么？

3.18 假设在程序中的某一点，如果累加器A中的值等于字符“Q”或“q”的ASCII码，那么8051就要跳转到标号EXI处去；如果不等于，8051继续执行下面的指令。要实现如上功能，相应的指令序列是什么？

3.19 假设指令

```
SJMP BACK
```

位于代码空间中地址0100H和0101H处，标号BACK代表位于地址00AEH处的指令。那么，上面这条指令所对应的机器语言代码的十六进制表示形式是什么?

3.20　假设指令

```
CJNE R7,#'Z',NOTZED
```

位于代码空间中地址022AH和022CH处，该指令所对应的机器语言代码的十六进制表示形式是什么?

3.21　指令

```
SETB 0D7H
```

的功能是什么?要实现同样的功能，更好的办法是什么?为什么?

3.22　以下两条指令有什么不同之处?

```
INC     A
INC     ACC
```

3.23　在如下指令中：

```
LJMP ONWARD
```

假设标号ONWARD代表位于代码空间中地址A0F6H处的指令，那么该指令所对应的机器语言代码是什么?

3.24　假设累加器*A*中的数据是5AH，那么执行

```
XRL A,#0FFH
```

指令后，累加器*A*中的数据又是多少？

3.25　假设累加器*A*中的数据是29H，那么执行


```
ORL A,#47H
```

指令后，累加器*A*中的数据变成多少?

3.26　假设执行

```
RLC A
```

指令前，程序状态字PSW中的数据为0C0H，累加器*A*中的数据为50H。在执行完上述指令后，累加器*A*中的数据又是多少?

3.27　假设执行

```
RRC A
```

指令前，程序状态字PSW中的数据为78H，累加器*A*中的数据为81H。执行指令后，累加器*A*中的数据又是多少?

3.28　假设8051的工作晶振是12MHz，P1.7的初始值是1，要想在引脚P1.7上生成一个持续时间为5μs的低电平脉冲，需要什么样的指令序列?

3.29　写一段程序，在引脚P1.0上生成一个频率为83.3kHz的方波（假设8051的工作晶振是 21MHz）。

3.30　写一段程序，每隔200μs，在引脚P1.7上生成一个持续时间为4μs的高电平脉冲。

3.31 写程序实现如图3-6所示的逻辑运算功能。对每一种情况，从输入发生变化，到输出发生变化，最坏情况下的传播延迟是多少?不妨设8051的工作晶振是12MHz。

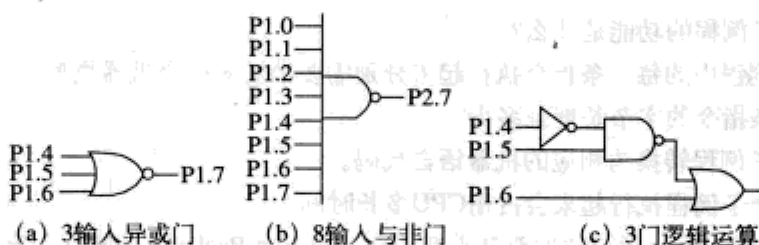


图3-6 用程序来实现逻辑功能

3.32 写程序实现如图3-7所示的逻辑运算功能。对每一种情况，从输入发生变化，到输出发生变化，最坏情况下的传播延迟是多少？不妨设8051的工作晶振是12MHz。

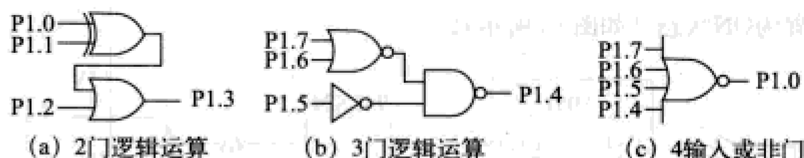


图3-7 用程序来实现逻辑功能

3.33 执行以下指令序列后，累加器A中的数据是多少?

```
MOV     A,#7FH
MOV     50H,#29H
MOV     R0,#50H
XCHD    A,@R0
```

3.34 指令

```
SETB P2.6
```

所对应的机器语言代码是什么?

3.35 什么指令可以把程序状态字PSW中的FLAG 0位复制到端口P1.5中?

3.36 在什么情况下，Intel的汇编器ASM51把泛型的JMP指令转换为LJMP指令?

3.37 假设8051中，在执行RET指令之前，内部RAM中部分单元的初值如下所示：

内部RAM地址	内容	特殊功能寄存器	内容
0B	9A	SP	0B
0A	78	PC	0200
09	56	A	55
08	34		
07	12		

那么在执行RET指令之后，程序计数器PC的值变成多少?

3.38 下面是一个8051的子例程：

```
SUB:    MOV   R0, #20H
LOOP:   MOV   @R0, #0
```

```
INC     R0
CJNE    R0, #80H, LOOP
RET
```

(a) 该子例程的功能是什么？

(b) 子例程中的每一条指令执行起来分别需要消耗多少个机器周期？

(c) 每条指令的字节长度是多少？

(d) 把子例程转换为相应的机器语言代码。

(e) 这个子例程执行起来会占用CPU多长时间？

3.39　如图3-8所示，一个4位的双列直插（Dual In-line Package，DIP）封装形式的开关和一个共阳极的7段LED被连接到8051上。在8051中会运行一个程序，连续不断地从DIP开关处读取4位数据，并将其以相应的十六进制数的形式显示在LED上。例如，如果读回的数据是1100B，那么，LED上会显示十六进制数C，因而，LED中的a到g这七段的状态分别是ON、OFF、OFF、ON、ON、ON和OFF。请注意，把8051的端口线设置为1，就会将相对应的LED段设置为ON状态（如图3-8所示）。

82

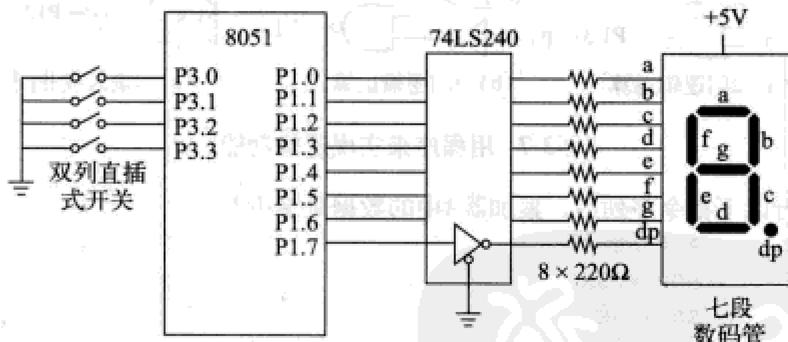


图3-8　DIP开关和七段数码管的接口

3.40　如果准备将一个数据送到下列存储器时，应该应用什么传送指令？

(a) 内部数据存储器

(b) 内部程序存储器

(c) 外部数据存储器

(d) 外部程序存储器

3.41　什么是查表？使用查表的好处是什么？

3.42　下面两条指令的区别是什么？ 详细说明每条指令的工作过程。

```
MOV   A,@R0
MOV   A,R0
```

3.43　找出下面的汇编语言程序所改变的存储单元，并推断出这些单元的最终内容。

```
        MOV   R0,#10H
REP:    MOV   @R0,#55H
        INC   R0
        CJNE  R0,#20H,REP
        MOV   R1,#20H
```

```
LOOP:   MOV   @R1,#AAH
        DEC   R1
        CJNE  R1,#5FH,LOOP
        END
```

3.44 找出下面程序所改变的存储单元，并给出这些单元的新数值。

```
        MOV   R0,#7FH
LOOP:   MOV   @R0,#0FFH
        DEC   R0
        CJNE  R0,#20H,LOOP
        MOV   R1,#30H
NEXT:   MOV   @R1,#00H
        INC   R1
        CJNE  R1,#5FH,NEXT
        END
```


3.45 假设地址为30H、78H和7FH的存储单元的初始值如下所示：

地址	内容
30H	55H
78H	00H
7FH	FFH

当执行下面的每条指令之后，这3个存储单元的内容变成了什么？为什么？（假设各条指令之间互不相干。）

(a) CPL　7FH

(b) CLR　78H

(c) MOV　7FH, 78H

3.46 编写汇编程序，实现将内部RAM的从30H到6FH存储单元的数值求和，并将计算结果存储到内部RAM的70H单元，要求写出每行代码的注释。

3.47 在数学里通常将阶乘运算用符号“!”来表示，阶乘在概率计算中应用很多。例如$5!=5\times4\times3\times2\times1$。编写汇编程序，计算存储在RAM的55H单元中的数值的阶乘，并将结果存放到RAM的77H单元。

3.48 累加器A中存放了1个4位的数据x（0除外）。编写程序利用查表方法得到$20\log10(x)$，并将计算结果进行四舍五入处理。

3.49 利用查表方法编写汇编程序，完成计算累加器A中的数据x的指数函数值$\exp(x)$。并要求将计算结果（通过舍入处理变成整数）的高字节和低字节分别存放到寄存器$R1$和$R0$中。例如在累加器A中存放的数据是2，那么计算结果应该是$R1=0$，$R0=7$。

3.50 编写汇编程序，实现将分别存放在$R1$和$R2$中的两个数据相乘。要求不能使用MUL AB指令，而是使用其他指令来完成这件事情。计算结果的高字节和低字节分别存放到$R3$和$R2$中。

3.51 什么是子例程？在汇编程序中使用子例程的好处是什么？

3.52 术语“嵌套子例程”是指两个或多个子例程结合使用，在其中一个子例程中调用其他子例程。

(a) 编写名为POW的子例程，用来计算存放在累加器*A*中数值的乘方，乘方的次数存放在寄存器*B*中。例如，如果*A*=2，*B*=3，那么结果为$2^3=8$。之后将计算结果的16位数据存放到累加器*A*（高字节）和寄存器*B*（低字节）中。

(b) 然后，再编写名为CALCULATE的子例程，用来计算3^4-2^3，并将16位的运算结果存放到累加器*A*（高字节）和寄存器*B*（低字节）中。要求通过调用子例程(a)中POW的方式完成求幂运算（即用一个数作为指数，另一个数作为底数，实现乘方运算）。

3.53 编写名为SUM的子例程，计算分别存放在累加器*A*和寄存器*B*中两个数值的加法运算，将计算结果存放到累加器*A*中。之后再编写一个名为TOTAL的子例程，计算分别存放于寄存器*R*1、*R*2和*R*3中的3个数值的总和。然后，再将TOTAL改成通过调用SUM子例程的方式来完成这3个数的求和运算。

3.54 8051提供了XCHD指令来完成两个数值的低4位的交换功能。编写一个名为XCHH的子例程，用来实现将两个分别存放于累加器*A*和寄存器*B*中的数值的高4位相互交换。

84

3.55 编写名为XRB的子例程，用来完成两个分别存放在*C*和*P*中的位数据的异或逻辑运算，并将最终结果存放到*C*中。要求在每条语句后都要加上注释。

3.56 下面的GUESSME子例程可用来完成一个非常有用的功能，而且是8051指令集所不具备的。

(a) 在每条语句后写出相应的注释。

(b) 你认为该程序能实现什么功能？

```
GUESSME:    CLR     C
            RRC     A
            DJNZ    R0,GUESSME
            END
```

3.57 编写汇编程序用来实现两个分别存放在*R*0和*R*1中数值的除法。将商存放到累加器*A*中，除数存放到寄存器*B*中。要求不能直接使用除法指令DIV AB。

3.58 在例3-27中，设计了XRB子例程以XRB C, P的格式完成两个位数据的异或逻辑运算。两个进行异或运算的位数据分别存放于*C*和*P*中，运算结果再存放到*C*中。按照以下两种方法重新编写此异或子例程。

(a) 将位数据插入到字节数据中，然后完成两个字节的异或运算。

(b) 使用JB和JNB指令。

3.59 从内部RAM的30H~39H单元分别存放着数字0~9，编写汇编程序反转这些数字的存储顺序，即0放到39H，1放到38H，等等。（提示：使用PUSH和POP指令。）

85

3.60 编写汇编程序，完成两个16位（双字节）数据的加法。

第4章　定时器操作

4.1　引言

本章将详细讨论8051的片上定时器，首先简单介绍一下广泛应用于微控制器或微处理器的定时器情况。

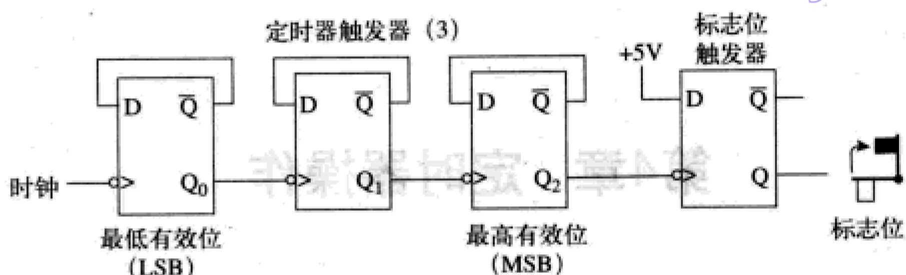
定时器实际上是一系列由时钟信号驱动的2分频触发器。时钟信号从第1个触发器输入，输出的信号频率为时钟信号频率的一半。第1个触发器输出信号作为第2个触发器的时钟，输出信号的频率同样减半，依此类推，信号频率每经过一级触发器就会减半，所以由n级触发器构成的定时器可以将时钟频率减为原来的$1/2^n$。最后一级触发器输出的信号驱动1个定时器溢出触发器，也称作标志位，其状态可以通过软件查询，也可以通过设置使得标志位被置位的时候触发一个中断。定时器触发器中的二进制数值是从定时器开始工作到当前时刻所计数的时钟脉冲（或"事件"）的个数。例如，一个16位计数器可以从0000H计数到FFFFH。当计数从FFFFH到0000H溢出时，溢出标志被置1）。

一个简单的3位定时器的工作过程如图4-1所示。定时器的每一级都是1个下降沿触发的D型2分频触发器（即$\overline{Q}$输出端与D输入端相连）。标志触发器就是一个简单的D型锁存器，由定时器的最后一级输出信号置位。图4-1b是定时器的时序图，从图中可以非常明显地看到第一级输出信号（$\overline{Q_0}$）翻转的频率是时钟频率的1/2，第二级是时钟频率的1/4，依此类推。定时器的计数采用十进制数表示，根据3位触发器的状态可以很容易地验证数值的正确与否。例如，当$Q_2=1$、$Q_1=0$、$Q_0=0$（$4_{10}=100_2$）时，计数值为4。

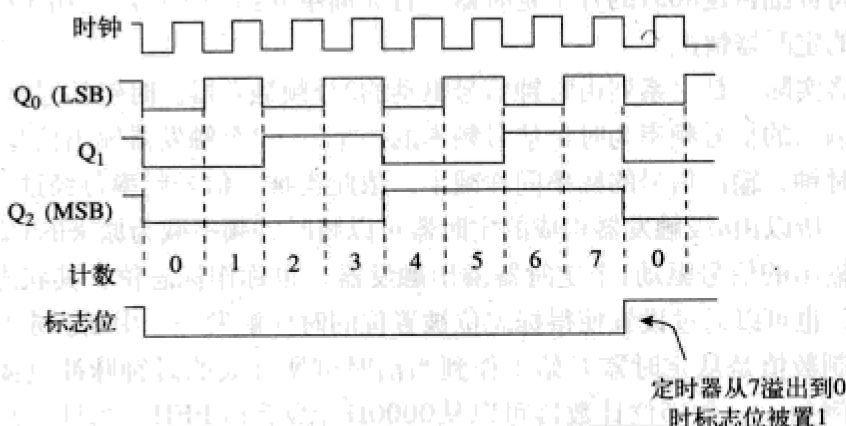
事实上，在所有面向控制的应用中都需要使用定时器，8051 也不例外。8051拥有两个16位定时器，每个定时器有4种工作模式。8052还增加了1个16位定时器，具有3种工作模式。定时器可用于定时、计数、波特率发生器。每个定时器都是16位的，因此，定时器的第16级（即最后1级）的输出频率是输入时钟频率的$1/2^{16}=1/65\,536$。

当使用中断定时器功能时，可以通过编程使定时器在特定的时间间隔发生溢出并置位溢出标志。溢出标志可用于引发程序执行某个特定的操作，如检查输入状态或者输出数据等。在某些应用中，也可以利用定时器准确的定时能力测量两种条件之间的时间间隔（例如脉冲宽度测量）。

计数模式用于测定某个事件发生的次数，而不是测量事件之间的时间间隔。任



(a) 结构图



(b) 时序图

图4-1 3位定时器

何使8051 IC某个引脚的电平由1变为0的外部激励都是一个“事件”。定时器还可为8051的串行端口提供波特率时钟信号。

8051的定时器可通过6个特殊功能寄存器来控制（见表4-1）。8052增加了5个特殊功能寄存器，用于控制第3个定时器。

表4-1 定时器专用寄存器

定时器SFR	用途	地址	可位寻址
TCON	控制	88H	是
TMOD	模式	89H	否
TL0	定时器0低字节	8AH	否
TL1	定时器1低字节	8BH	否
TH0	定时器0高字节	8CH	否
TH1	定时器1高字节	8DH	否
T2CON*	定时器2控制	C8H	是
RCAP2L*	捕获定时器2低字节	CAH	否
RCAP2H*	捕获定时器2高字节	CBH	否

(续)

定时器SFR	用 途	地 址	可位寻址
TL2 *	定时器2低字节	CCH	否
TH2 *	定时器2高字节	CDH	否

* 8032/8052所特有。

4.2 定时器模式寄存器 (TMOD)

TMOD寄存器分为两组各4位，分别用于设置定时器0和定时器1的工作模式（如表4-2和表4-3所示）。

表4-2 TMOD（定时器模式）寄存器简表

位	名 称	定 时 器	描 述
7	GATE	1	门控位。当被置为1时，定时器只有在为高电平时才开始工作
6	C/$\overline{T}$	1	计数器/定时器选择位 1=计数器 0=定时器
5	M1	1	模式位1(见表4-3)
4	M0	1	模式位0(见表4-3)
3	GATE	0	定时器0的门控位
2	C/$\overline{T}$	0	定时器0的计数器/定时器选择位
1	M1	0	定时器0的M1位
0	M0	0	定时器0的M0位

表4-3 定时器模式

M1	M0	模 式	描 述
0	0	0	13位定时器模式(8048模式)
0	1	1	16位定时器模式
1	0	2	8位自动重载模式
1	1	3	分立定时器模式 定时器0：TL0是由定时器0的模式控制位控制的8位定时器，TH0除了受定时器1的模式控制位控制外，其余和TL0相同 定时器1：停止使用

TMOD寄存器不可位寻址，事实上也没有这个必要。通常情况下，在程序开始运行的时候由初始化TMOD来选择定时器的工作模式。此后，通过访问定时器的其他特殊功能寄存器可以使定时器执行启动、停止等操作。

4.3 定时器控制寄存器 (TCON)

TCON 寄存器包括定时器0和定时器1的状态位和控制位（如表4-4所示）。

TCON的高4位（TCON.4~TCON.7）用于控制定时器的启动和停止（TR0，TR1），
89
或用来标志定时器的溢出 (TF0，TF1)。本章中的各个例子将会经常用到这些位。

表4-4 TCON(定时器控制)寄存器简表

位	符号	位地址	描述
TCON.7	TF1	8FH	定时器1溢出标志位。溢出时由硬件置1，软件清零或者在微处理器转向中断服务程序时由硬件自动清零
TCON.6	TR1	8EH	定时器1运行控制位。由软件置位或清除定时器的启动或停止
TCON.5	TF0	8DH	定时器0溢出标志位
TCON.4	TR0	8CH	定时器0运行控制位
TCON.3	IE1	8BH	外部中断1边沿触发标志。当检测到引脚上出现信号的下降沿时，由硬件置位；本标志位由软件清除，或在CPU转向中断服务程序时由硬件清除
TCON.2	IT1	8AH	外部中断1触发方式控制位，通过软件置位/清除来实现由下降沿/低电平触发外部中断：TI1＝1下降沿触发，TI1＝0低电平触发
TCON.1	IE0	89H	外部中断0边沿触发标志
TCON.0	IT0	88H	外部中断0触发方式控制位

TCON的低4位（TCON.0~TCON.3）与定时器无关。它们用于检测和初始化外部中断。在第6章关于中断的讨论中将会对这4位加以介绍。

4.4 定时器模式和溢出标志

下面对每个定时器分别进行讨论。因为8051有两个定时器，所以文中用*x*符号指代定时器0和定时器1。例如，对于不同定时器，TH*x*代表TH1或TH0。

90
每种模式下定时器寄存器TL*x*、TH*x*以及溢出标志TF*x*的设置如图4-2所示。

4.4.1 13 位定时器模式 (模式0)

模式0是13位定时器模式，它与8051的前身8048兼容，在现在的新设计中已经很少使用（见图4-2a）。定时器高字节(TH*x*)与低字节(TL*x*)的低5位共同构成1个13位
91
定时器。低字节(TL*x*)的高3位未使用。

4.4.2 16位定时器模式（模式1）

模式1是16位定时器模式，除了工作在16位以外与模式0没有什么不同。时钟信号输入到定时器寄存器的高低字节(TL*x*/TH*x*)，定时器的计数随着接收到的时钟脉冲增加：0000H，0001H，0002H，等等。当计数值从FFFFH回到0000H时，计数器发生溢出，定时器溢出标志被置为1，溢出后定时器会继续计数。溢出标志位于TCON的TF*x*位，可用软件读写（如图4-2b所示）。

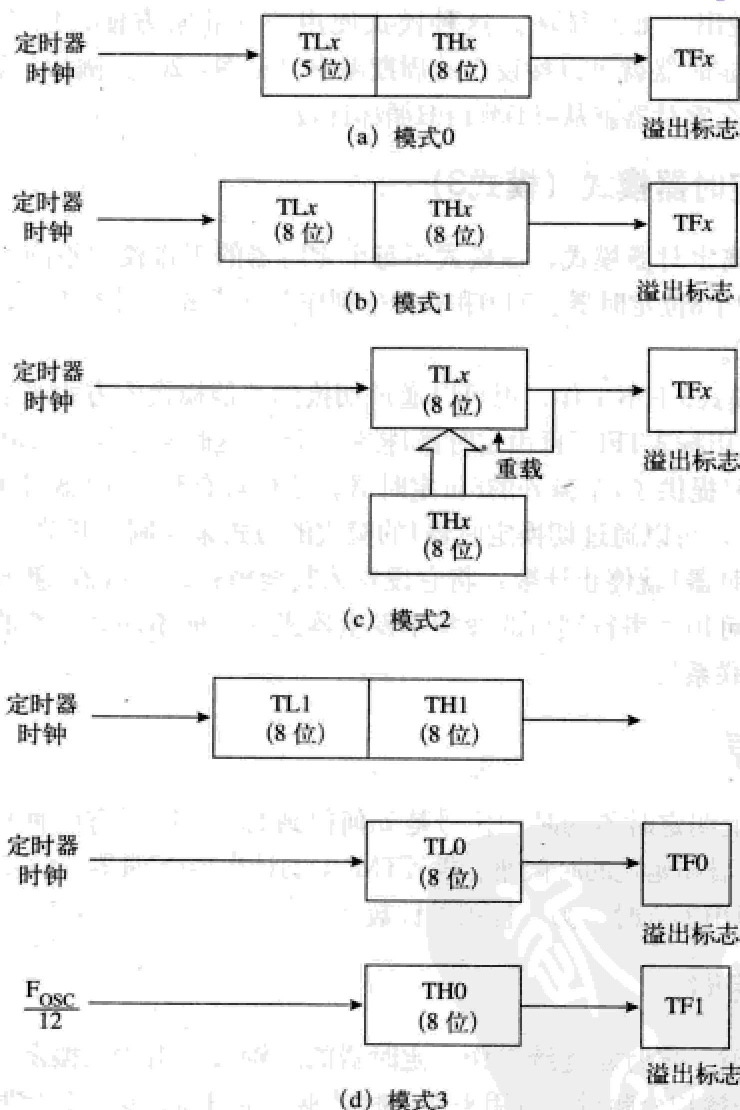


图4-2 定时器模式

定时器寄存器数值的最高有效位（MSB）是TH*x*的第7位，最低有效位（LSB）是TL*x*的第0位。最低有效位以输入时钟频率的1/2速率翻转，同时，最高有效位的翻转速率为时钟频率的1/65 536（即2^{16}）。软件可以在任何时候读写定时器寄存器（TL*x*/TH*x*）。

4.4.3 8位自动重装载模式（模式2）

模式2是8位自动重装载模式。这时定时器的低字节（TL*x*）作为1个8位定时器工作，高字节（TH*x*）则用于保存重载值。当计数值超出FFH时，不仅定时器溢出标志置1，同时TH*x*中保存的值被载入到TL*x*。计数从载入的数值开始直到下一个

FFH再次产生溢出，如此循环。这种模式使用起来非常方便，因为一旦设置好TMOD和TH*x*，定时器就可以按设定的周期溢出（见图4-2c）。例如，如果TH*x*中的数值是4FH，那么定时器就从4FH到FFH循环计数。

4.4.4 分离定时器模式（模式3）

模式3是分离定时器模式，该模式下每个定时器的工作模式不同。在模式3下，定时器0分离为2个8位定时器。TL0和TH0分别作为独立的定时器工作，对应的溢出标志是TF0和TFl。

定时器l在模式3下不工作，但可以通过切换到其他模式的方式来启动它。唯一的限制是这时溢出标志TFl不再由定时器1控制，因为这时它已经与TH0相关联。

模式3为8051提供了1个额外的8位定时器，这样就有3个定时器可用。当定时器0工作在模式3时，可以通过切换定时器1的模式的方式来控制其开关。将定时器1设置为模式3，定时器1就停止计数；将它设置为其他模式，定时器1就开始计数。这样，定时器1仍可用作串行端口的波特率发生器或在其他不需要中断的场合（因为它不再和TF1相联系）。

4.5 时钟源

图4-2没有说明定时器的时钟信号是如何得到的。定时器有两种时钟信号源可供选择，在定时器初始化的时候通过设置TMOD的计数器/定时器（C/$\overline{\text{T}}$）位来选择。一种时钟信号源用于定时，另一种用于计数。

4.5.1 中断定时

当C/$\overline{\text{T}}$＝0时，定时器连续工作，定时器的时钟信号由片上振荡器提供。振荡器提供的信号先被12分频后，再用来作为时钟驱动定时器，这样，定时器的时钟信号频率被降至一个对大部分应用来说较为合理的水平。

92

当处于连续工作模式时，定时器用于**中断定时**。定时器寄存器（TL*x*/TH*x*）的数值以片上振荡器频率的1/12的速率增加。因此，12MHz的晶振可产生1MHz的驱动时钟信号。定时器经过固定数目的时钟脉冲后溢出，该数值取决于定时器寄存器（TL*x*/TH*x*）的初始值。

4.5.2 事件计数

如果C/$\overline{\text{T}}$＝1，定时器由外部信号源提供计数脉冲。大多数应用中，每发生1个“事件”，外部信号源向定时器发送1个脉冲，引发定时器执行**事件计数**操作。由于每发生1个事件，定时器寄存器中的16位计数值就加1，所以通过软件读定时器的寄

存器（TLx/THx）的值就可知道事件发生的次数。

外部时钟信号通过端口3的引脚输入，这是它们的第二功能。端口3的第4位（P3.4）是定时器0的外部时钟信号输入端，记作T0。P3.5是定时器1的时钟信号输入端，记作T1（如图4-3所示）。

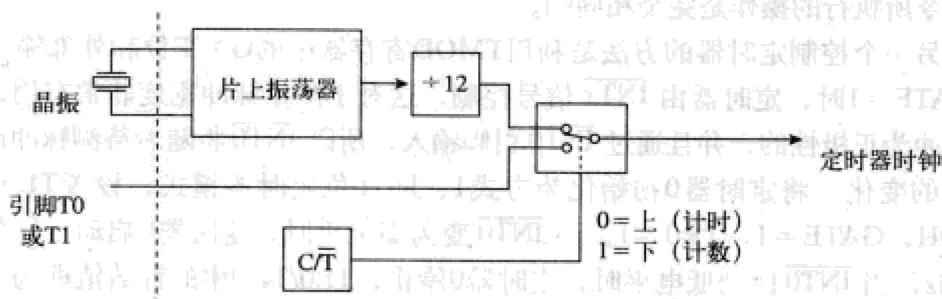


图4-3 时钟信号源

在计数应用中，一旦外部输入信号（Tx）发生由1到0的跳变，定时器寄存器的值就加1。系统在每个机器周期的S5P2期间对外部输入进行采样，如果输入信号在某个周期中为高电平，而且在随后的下一个周期变为低电平，那么计数值加1。如果检测到输入信号的这种跳变，那么在下一个周期的S3P1期间，更新定时器寄存器的数值。由于系统识别1个从1到0的跳变需要2个机器周期（2μs），可识别的外部信号的最高频率为500kHz（假设时钟频率是12MHz）。

4.6 定时器的启动、停止和控制

图4-2描述了定时器寄存器TLx、THx和溢出标志TFx的各种配置关系。图4-3则给出了2种可能的驱动定时器的时钟信号源。下面将研究如何启动、停止和控制定时器。

启动和停止定时器的最简单方法是用软件设置TCON寄存器中的运行控制位THx。系统复位后TRx被清零，所以在默认情况下定时器是停止的。可以利用软件将TRx置1来启动定时器（如图4-4所示）。

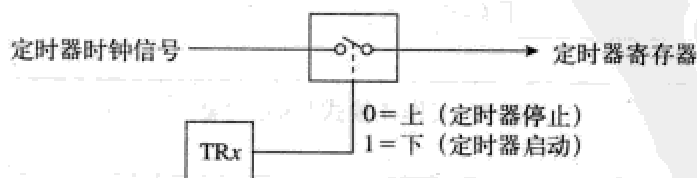


图4-4 定时器的启动和停止

TRx所在的TCON寄存器是可位寻址的，因此，在程序中很容易利用相应的指令来启动和停止定时器。例如，可用下面的指令启动定时器0：

```
SETB TR0
```

可用下面的指令停止定时器0：

```
CLR TR0
```

汇编器在编译时会将符号TR0转换为相应的位地址。所以SETB TR0和SETB 8CH两条指令所执行的操作是完全相同的。

另一个控制定时器的方法是利用TMOD寄存器中的GATE位和外部输入 $\overline{\text{INT}x}$ 。当GATE＝1时，定时器由 $\overline{\text{INT}x}$ 信号控制。这对于测量脉冲宽度非常有用，假设待测脉冲为正极性的，并且通过 $\overline{\text{INT0}}$ 引脚输入，所以 $\overline{\text{INT0}}$ 将随着待测脉冲产生高低电平的变化。将定时器0初始化为方式1，即16位定时器模式，设置TL0/TH0 ＝ 0000H，GATE＝1，TR0＝1。当 $\overline{\text{INT0}}$ 变为高电平时，定时器0启动，计数频率为1MHz，当 $\overline{\text{INT0}}$ 回到低电平时，定时器0停止，TL0/TH0中的计数值即为以微秒为单位的接于 $\overline{\text{INT0}}$ 的待测脉冲宽度（可通过软件设置使得在信号回到低电平的同时触发1个中断）。

94

在图4-5中描述了定时器1工作在模式1即16位定时器模式的完整控制逻辑关系。图中表示出了定时器寄存器TL1/TH1和溢出标志TF1，同时给出了可能的时钟信号源和定时器的启动、停止和控制部分。

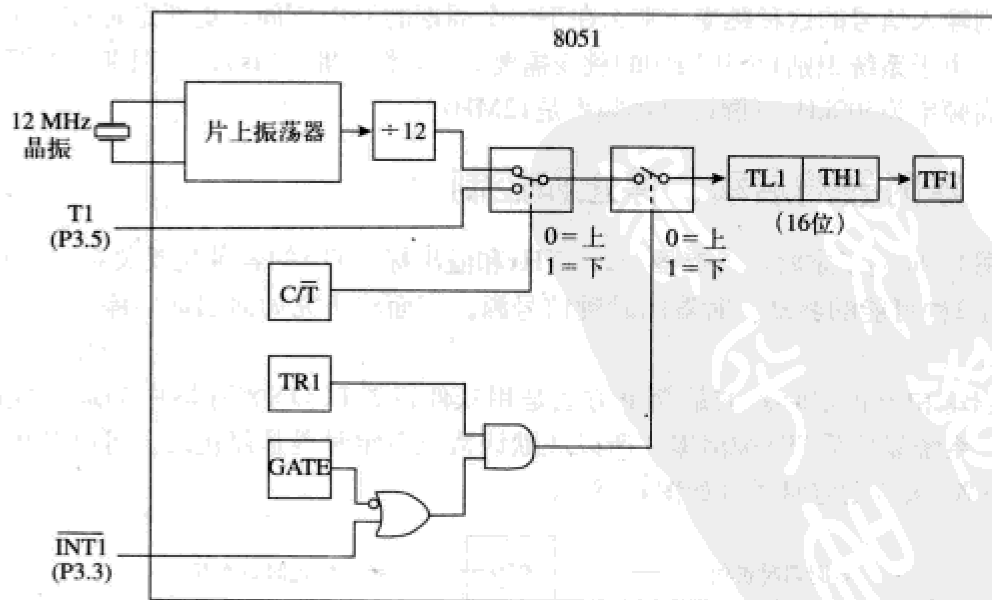


图4-5 工作在模式1的定时器1

例4-1 **图4-5中，定时器1工作于模式1。列出图中所示的定时器寄存器和控制位，给出它们的位地址或字节地址，并列出各控制位所在的特殊功能寄存器。**

答案：

定时器寄存器：

TH1，字节地址8DH；

TL1，字节地址8BH。

定时器控制/模式选择位：

TR1，位地址8EH（在TCON中）；

TF1，位地址8FH（在TCON中）。

C/$\overline{T}$，TMOD（字节地址88H）的第6位；

GATE，TMOD（字节地址88H）的第7位。

讨论：在列出的4个定时器控制/模式选择位中，只有TR1和TF1是可位寻址的。可以直接对这些位进行置位或清除操作，以启动或停止定时器。也可以在需要的时候查询它们的状态。C/$\overline{T}$位和GATE位通常只在程序开始处设置1次，用于选择定时器的工作模式。

4.7 定时器寄存器的初始化和访问

通常情况下，只在程序开始处对定时器进行1次初始化，以设置正确的工作模式。在程序当中，可以根据实际应用的需要启动、停止定时器，检测、清除溢出标志，读取或更新定时器寄存器，以及其他操作。

TMOD是首先被初始化的寄存器，因为需要通过它来设置定时器的工作模式。例如，下面的指令将定时器1初始化为16位定时器（模式1），由片上振荡器提供时钟信号（中断计时）：

```
MOV   TMOD,   #00010000B
```

这条指令的作用是设置M1＝0、M0＝1，选择工作方式1，置C/$\overline{T}$＝0、GATE＝0，选择定时模式，同时清除定时器0的模式位（见表4-2）。当然，在运行控制位TR1被置1前，定时器不会开始计时。

如果程序需要1个计数初值，寄存器TL1/TH1也必须被初始化。定时器的计数从初始值开始直到FFFFH，然后在FFFFH到0000H的跳变发生时置位溢出标志，如果要定时100μs，可以置TL1/TH1的初值为0000H减去100，正确的数值是−100或FF9C。下面的指令可以完成上面的计时工作：

```
MOV   TL1, #9CH
MOV   TH1, #0FFH
```

下面这条指令置运行控制位TR1为1，启动定时器1：

```
STEB   TR1
```

溢出标志在100μs后被自动置1。软件可以在等待循环中利用条件分支指令不断查询定时器1溢出标志的状态，如果为0则继续检测，直到定时器1溢出标志被置1才跳出

等待循环：

```
WAIT:      JNB   TF1, WAIT
```

当定时器溢出后，需要通过软件停止定时器1并清除溢出标志：

```
CLR   TR1
CLR   TF1
```

即时读取定时器

在某些应用中，需要即时读取定时器寄存器中的数值。在这样的场合，有一个潜在的问题，不过可以通过软件很容易地避免。由于需要分别读取2个定时器寄存器（即THx和TLx）的内容，因此，如果在这2次读取操作过程中发生了低字节向高字节的进位，那么就会出现“相位误差”，读取到的数值可能已不再存在。针对此问题可采取的解决方案是：先读取高字节，再读取低字节，然后再读取1遍高字节。如果两次读取的高字节内容不同，那么重复读取操作。下面的指令把定时器寄存器TL1/TH1中的内容分别读取到寄存器R6/R7中，同时有效地解决了上述问题：

```
AGAIN: MOV      A,TH1
       MOV      R6,TL1
       CJNE     A,TH1,AGAIN
       MOV      R7,A
```

4.8 短、中、长定时间隔

8051可以定时的时间间隔有多长？在讨论这个问题之前，首先假设8051由12MHz晶振驱动，片上振荡器的时钟信号经过12分频后驱动定时器，频率为1MHz。

8051能够计时的最短时间间隔是有限的，它不取决于定时器的时钟频率，而取决于软件。可以推测的是，在特定的间隔内必须发生点什么，因此，指令执行所消耗的时间决定了最短间隔的大小。8051最短的指令执行时间为1个机器周期，即1μs。

96

表4-5列出了产生不同长度时间间隔的方法（假设8051工作于12 MHz晶振频率下）。

表4-5 产生不同长度时间间隔（工作于12MHz）的方法

最长时间间隔（微秒）	方　法
≈10	软件编写
256	8位定时器，自动重载模式
65 536	16位定时器
无限长	16位定时器及软件循环

例4-2 脉冲波形的产生

编写1个脉冲波形产生程序，在引脚P1.0产生最高频率的周期性脉冲波。能产生的最高频率是多少？该周期性脉冲波形的占空比是多少？

答案：

```
8100          5          ORG    8100H
8100 D290     6   LOOP:  SETB   P1.0 ;one machine cycle
8102 C290     7          CLR    P1.0 ;one machine cycle
8104 80FA     8          SJMP   LOOP ;two machine cycles
              9          END
```

讨论： 该程序在引脚P1.0上产生了周期为4μs、频率为250 kHz的脉冲波形。在每个周期中，信号为高电平的时间是1μs，为低电平的时间为3μs，占空比为1/4＝0.25，即25%（如图4-6所示）。

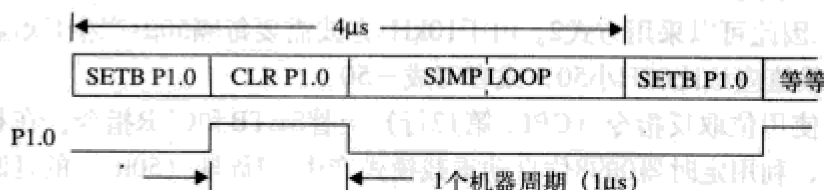


图4-6 例4-2的波形

乍一看，图4-6中指令的位置似乎有错，但事实上并非如此。例如，指令SETB P1.0，在其执行的最后阶段，也就是S6P2期间，才把P1.0设置为1。

在程序的循环体中插入NOP指令可以延长脉冲的周期。每条NOP指令可以使脉冲周期增加1个机器周期，即1μs。例如，在指令SETB P1.0之后增加2条NOP指令可以使输出波形变为方波（占空比＝50%），周期变为6μs，频率变为166.7 kHz。当脉冲周期长度超过一定范围之后，用软件调整实现的方式就很麻烦，这时建立延时的最好办法是使用定时器。

例4-3 方波的产生

编写程序，在引脚P1.0产生尽可能高频率的方波信号。最高频率和占空比是多少？

答案：

```
8100           5          ORG 8100H
8100 B290      6   LOOP:  CPL  P1.0     ;one machine cycle
8102 80FC      7          SJMP LOOP     ;two machine cycles
               8          END
```

讨论： 周期为6μs，高电平时间＝低电平时间＝3μs。频率为166.67kHz，占空比为50%，是一个方波，而例4-2不是方波。

用8位自动重载模式（即模式2）可以方便地获得中等长度的时间间隔。由于是8位计数，定时器溢出前能够产生的最长定时间隔为2^8＝256μs。

例4-4 10 kHz方波

编写程序，利用定时器0在引脚P1.0产生10 kHz的方波。

答案：

```
8100                6          ORG   8100H
8100  758902        7          MOV   TMOD,#02H  ;8-bit auto-reload mode
8103  758CCE        8          MOV   TH0,#-50   ;-50 reload value in TH0
8106  D28C          9          SETB  TR0        ;start timer
8108  308DFD        10  LOOP:  JNB   TF0,LOOP   ;wait for overflow
810B  C28D          11         CLR   TF0        ;clear timer overflow flag
810D  B290          12         CPL   P1.0       ;toggle port bit
810F  80F7          13         SJMP  LOOP       ;repeat
                    14         END
```

讨论：上面的程序在P1.0上产生了方波，高低电平的时间均为50μs。由于时间间隔小于256μs，因此可以采用方式2。由于10kHz方波需要每隔50μs产生1次溢出，因此，TH0中的重载值应该比00H小50，或者写成－50。

程序中使用位取反指令（CPL，第12行）代替SETB和CLR指令。在相邻两次取反操作之间，利用定时器0的8位自动重载模式产生1/2周期（50μs）的延时。这里重载值没有用十六进制表示，而是用十进制的－50表示（第8行），汇编器会进行相应的转换。注意，每次溢出后软件都会清除定时器0的溢出标志位（TF0）（第11行）。

倘若要产生大于256μs的定时间隔，就需要使用16位定时器模式，即模式1。在该模式下，能够产生的最长的延时为$2^{16}=65\ 536$ μs，大约0.066s。但每次溢出后要重新设置定时器寄存器的初始值，这是使用模式1的不便之处，而在模式2中，重装载过程是自动完成的。

98

例4-5　1kHz方波

编写程序，利用定时器0在引脚P1.0产生1kHz的方波。

答案：

```
8100                6          ORG   8100H
8100  758901        7          MOV   TMOD,#01H  ;16-bit timer mode
8103  758CFE        8   LOOP:  MOV   TH0,#0FEH  ;-500 (high byte)
8106  758A0C        9          MOV   TL0 #0CH   ;-500 (low byte)
8109  D28C          10         SETB  TR0        ;start timer
810B  308DFD        11  WAIT:  JNB   TF0,WAIT   ;wait for overflow
810E  C28C          12         CLR   TR0        ;stop timer
8110  C28D          13         CLR   TF0        ;clear timer overflow
                                                 flag
8112  B290          14         CPL   P1.0       ;toggle port bit
8114  80ED          15         SJMP  LOOP       ;repeat
                    16         END
```

讨论：1kHz方波的高低电平时间都为500μs。由于这个时间间隔超过了256μs，不能使用模式2，而需要使用16位定时器模式（即模式1）。和模式0相比，软件的差别在于每次溢出之后需要重新初始化定时器寄存器TL0和TH0（第8行和第9行）。

上面的程序输出的脉冲频率与1kHz略有差别。引起差异的原因如下：在溢出后，需要执行若干指令来重新初始化定时器寄存器。如果需要得到精确的1kHz方波，TL0/TH0的重载值需要做微小的调整。在自动重载模式中则不会出现这样的问题，因为定时器溢出后直接回到TH0中的重载值，其间没有停顿。

由于8051中有两个定时器，所以程序可以在各自的输出引脚产生不同的波形。

例4-6 同时使用两个定时器

编写程序，在引脚P1.0上产生频率为10kHz的方波，同时，在引脚P2.0产生频率为1kHz的方波。

答案：

```
8100            6           ORG 8100H
8100 758912     7           MOV TMOD, #12H   ;timer 1 in mode 1
                8                            ; timer 0 in mode 2
8103 758CCE     9           MOV TH0, #-50    ;-50 reload value
                10                           ; in TH0
8106 D28C       11          SETB TR0         ;start timer 0
8108 758DFE     12   LOOP:  MOV TH1, #0FEH   ;-500 (high byte)
810B 758B0C     13          MOV TL1, #0CH    ;-500 (low byte)
810E D28E       14          SETB TR1         ;start timer 1
8110 308D04     15   WAIT:  JNB TF0, NEXT    ;timer 0 overflow?
                16                           ; no: check timer 1
8113 C28D       17          CLR TF0          ;yes: clear timer 0
                18                           ; overflow flag
8115 B290       19          CPL P1.0         ;toggle P1.0
8117 308FF6     20   NEXT:  JNB TF1, WAIT    ;timer 1 overflow?
                21                           ; no: check timer 0
811A C28E       22          CLR TR1          ;stop timer 1
811C C28F       23          CLR TF1          ;clear timer 1
                24                           ; overflow flag
811E B2A0       25          CPL P2.0         ;toggle P2.0
8120 80E6       26          SJMP LOOP        ;repeat
                27          END
```


讨论：本例利用定时器0和定时器1在引脚P1.0和P2.0产生两个方波。两个定时器的工作模式在初始化TMOD寄存器的过程中同时设定。即使两个定时器同时运行，对各自溢出标志位的检测也要依次进行。定时器0需要首先检测，因为其产生的10kHz方波的周期较短，这样可以防止漏检定时器0的每次溢出事件。需要注意，这里的定时器0被设置成了模式2也就是自动重装载模式，故在每次溢出发生之后无需软件对其再次赋初值。由于工作在模式1，所以定时器1需要在每次溢出之后对其装载初值。

例4-7 蜂鸣器接口

1个蜂鸣器连接在引脚P1.7上，1个反跳开关连接引脚P1.6（如图4-7所示）。编写程序，监测开关的逻辑电平，当电平发生1到0的跳变时，蜂鸣器鸣响1s。

答案：

```
0064              6   HUNDRED   EQU    100      ;100 × 10000 us=1 sec.
D8F0              7   COUNT EQU         -10000
8100              8             ORG     8100H
8100    758901    9         MOV TMOD, #01H      ;use timer 0 in mode 1
8103    3096FD    10  LOOP: JNB P1.6,LOOP       ;wait for 1 input
8106    2096FD    11  WAIT: JB  P1.6,WAIT       ;wait for 0 input
8109    D297      12            SETB    P1.7    ;turn buzzer on
```

100

```
810B  128112  13            CALL  DELAY     ;wait 1 second
810E  C297    14            CLR   P1.7      ;turn buzzer off
8110  80F1    15            SJMP  LOOP
              16  ;
8112  7F64    17  DELAY:    MOV   R7,#HUNDRED
8114  758CD8  18  AGAIN:    MOV   TH0,#HIGH COUNT
8117  758AF0  19            MOV   TL0,#LOW COUNT
811A  D28C    20            SETB  TR0
811C  308DFD  21  WAIT2:    JNB   TF0,WAIT2
811F  C28D    22            CLR   TF0
8121  C28C    23            CLR   TR0
8123  DFEF    24            DJNZ  R7,AGAIN
8125  22      25            RET
              26            END
```

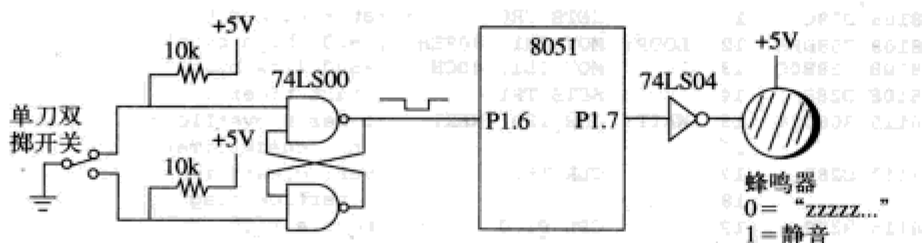


图4-7　蜂鸣器的例子

讨论：图4-7中的蜂鸣器是1个压电陶瓷转换器，以直流电压驱动时可以产生振动。典型的产品是Projects Unlimited公司的AI-430，以5V直流电压驱动能够发出频率为3kHz的音调。由于AI-430需要7 mA的电流，电路中用1个反相器（74LS04）作为驱动。由于8051端口1的引脚最大可吸收1.6 mA的电流，所以这个反相器是必须用的。购买1个AI-430需要几美元。

软件的主循环由6条指令构成（第10行到第15行）。第10行的指令是单指令循环，等待P1.6的输入电平变为高。第11行的指令又是单指令循环，等待输入电平跳变到低电平。当跳变发生时，蜂鸣器鸣叫1s。这个工作是由随后的3条指令实现的：首先，P1.7被置1，使蜂鸣器鸣叫（第12行）；接着调用延时子例程，延时1s（第13行）；然后清除P1.7，关闭蜂鸣器（第14行）；最后在程序标号LOOP处再次执行该循环（第15行）。

延时子例程（第17行到第25行）使用了表4-6中的“16位定时器及软件循环”方法。理论上，用这种方法可以产生任意长度的延时。在本例中，利用16位模式并置TH0/L0为−10 000可以产生10 000μs的延时。再利用*R*7作为计数器，将这段10 000μs的延时程序（第18行到第23行）重复执行100次，就得到了1s的延时。

101

上面的例子没有考虑到两种情况：（1）如果在蜂鸣器鸣叫的1s时间内输入信号发生翻转，由于此时软件正在运行延时子例程，无法检测到这个跳变。（2）如果输入信号跳变得非常快，短于1ms，那么JNB和JB指令都无法检测到。第1个问题在本章最后的问题5中解决。第2个问题只能利用中断解决，在发生1到0的跳变时利用中

断将跳变信息锁存到1个状态标志位中去。关于中断的内容将在第6章中讨论。

4.9 精确频率的产生

前面几节中所讨论的通过定时器产生方波输出的例子中，如果仔细分析，就频率指标而言实际上是存在一些误差的。误差的来源有二个：一是在计算计数初值时产生的舍入误差；二是实现定时器定时功能的指令本身的执行需要耗费一定的时间。

4.9.1 舍入误差的消除

在期望波形的每个周期所包含的定时器计数周期的数目是非整数的情况之下，会出现舍入误差。由于这个数值最终要写入到8051定时器的寄存器中，所以必须将其四舍五入到一个整数。

例4-8 舍入误差

假设欲产生一个3kHz的方波，定时器的重载值是多少？计算产生的舍入误差，之后选择合适频率的晶振消除该舍入误差。

102

答案：3kHz方波的周期是333.33μs，其高电平时间＝低电平时间＝166.67μs。由于8051只能处理整数形式的计数初值，所以在这种情况下定时器的重载值应该比溢出值低167，或者−167。注意，此处我们将计数总个数四舍五入到了整数。由此而产生的实际周期为167×2×1μs＝334μs，也就是说实际频率为2.994kHz，其舍入误差为：

$$舍入误差=\frac{|f_{desired}-f_{rounded\text{-}off}|}{f_{desired}}\times 100\%$$

$$=\frac{|3kHz-2.994kHz|}{3kHz}\times 100\%=0.2\%$$

讨论：从上面的计算可知，如果使用12MHz的晶振，将产生舍入误差。假设计数个数还取167，我们的目的是计算产生精确的3kHz方波所需要的晶振的频率。

期望周期＝333.33μs，因此：

期望高电平时间＝期望低电平时间

＝166.67μs

＝两次相邻溢出事件发生的时间间隔

＝167×机器周期

因此，机器周期＝166.67μs/167＝0.9980239μs

与机器对应的频率＝1/0.9980239μs＝1.00198MHz

所以，要想产生3kHz的精确频率，需要的理想晶振频率＝1.00198MHz×12＝12.0238MHz。

4.9.2 指令执行所耗时间的补偿

程序指令的执行是需要一定的时间才能完成的。最简单的指令需要1个机器周期，最复杂的需要4个机器周期。因此如果我们想获得精准的频率指令，执行造成的这部分时间延迟也需要考虑的，需要在初始化定时器时通过调整计数初值的方式将这部分时间延迟补偿回来。下面的例4-9就是这方面的一个简单示范。

例4-9 重新改写例4-5的程序，要求补偿掉由于相关指令执行所造成的时间延迟。

答案：

```
8100                 1            ORG 8100H
8100 758901          2            MOV TMOD, #01H   ;16-bit timer mode
8103 758CFE          3    LOOP:   MOV TH0, #0FEH   ;-490 (high byte)
8106 758A0E          4            MOV TL0, #0EH    ;-490 (low byte)
8109 D28C            5            SETB TR0         ;start timer
810B 308DFD          6    WAIT:   JNB TF0, WAIT    ;wait for overflow
810E C28C            7            CLR TR0          ;stop timer
8110 C28D            8            CLR TF0          ;clear timer overflow flag
8112 B290            9            CPL P1.0         ; toggle port bit
8114 80ED            10           SJMP LOOP        ; repeat
                     11           END
```

讨论：该程序除了重载值和例4-5的程序有所区别外，其余部分是完全相同的。为了补偿指令执行所造成的时间延迟，重载值从−500变成了−490，下面我们来分析一下为什么如此取值。

在软件的循环体中包含了2条MOV指令，紧跟着是SETB、JNB、2条CLR、CPL和SJMP指令。每条MOV、JNB和SJMP指令需要2个机器周期，而每条SETB、CLR和CPL指令需要1个机器周期。

P1.0的初始状态可能是高电平也可能是低电平，此处假设其为低电平。头2条MOV指令每条需要2个机器周期的执行时间，而SETB需要1个机器周期。接着的JNB指令的作用是决定定时间隔的长度，将被反复执行一直到定时器0发生溢出（溢出标志位TF0被置1，跳出循环），每次执行JNB指令需要花费2个机器周期。

当定时器0发生溢出的时候，P1.0的电平状态应该立刻反转。但是，在本程序中，需要延迟3个机器周期才最终实现了P1.0的电平反转，这额外的延迟是由于指令执行所耗费的CPU时间造成的。在图4-8中明确标识出了P1.0端口和TR0，详细描述了这一过程。需要注意的是，P1.0引脚输出方波的高电平持续时间和低电平持续时间是由先后两次执行CPL指令的时间间隔决定的。图4-8表明该过程所需要的时间为：一个完整的定时周期（490μs），溢出之后的CLR和CPL指令执行时间（3μs），还有定时器0被下一次重新启动之前执行SJMP、MOV和SETB指令所需要的时间（7μs）。

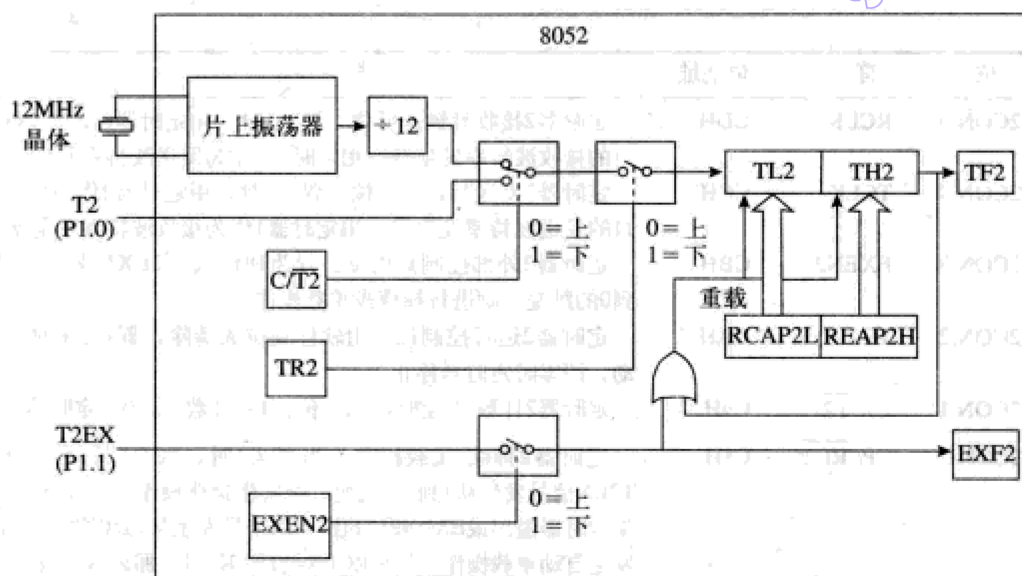


图4-8 定时器2的16位自动重装载模式

所以，各部分时间的总和是500μs，也就是说高电平和低电平的持续时间都是500μs，于是在P1.0引脚产生了精确的1kHz方波。

4.10 8052的定时器2

在8052 IC中增加的第3个定时器是前面刚刚讨论过的两个定时器的有力补充。此前在表4-1中已经提到，8052有5个额外的特殊功能寄存器供定时器2使用，包括定时器寄存器TL2和TH2、控制寄存器T2CON、捕获寄存器RCAP2L和RCAP2H。

定时器2的工作模式由其控制寄存器T2CON（见表4-6）决定。与定时器0和定时器1一样，定时器2可以作为定时器也可作为计数器使用。其时钟信号可由片上振荡器提供，或通过引脚T2由外部提供。T2是8052端口1的第0位（P1.0），这是其第二功能。T2CON的C/$\overline{T}_2$位决定了是使用外部时钟信号还是内部时钟信号，与定时器0和定时器1的TCON寄存器的C/$\overline{T}$位功能相同。不考虑时钟信号源的差别，定时器2有3种工作模式：自动重载模式、捕获模式和波特率发生器。

表4-6 T2CON（定时器2控制）寄存器简表

位	符	位地址	描 述
T2CON.7	TF2	CFH	定时器2溢出标志（当TCLK或RCLK＝1时，定时器溢出不会将TF2置1）
T2CON.6	EXF2	CEH	定时器2外部标志。当EXEN2＝1，且T2EX信号发生由1到0的跳变引发捕获或重载时，EXF2被置1；当定时器中断被启用时，EXF2＝1会触发中断，使CPU转向中断服务程序，此标志位由软件清除

（续）

位	符	位地址	描述
T2CON.5	RCLK	CDH	定时器2接收时钟选择位。置为1时，用定时器2作为串行端口的接收波特率发生器，用定时器1作为发送波特率发生器
T2CON.4	TCLK	CCH	定时器2发送时钟选择位。置为1时，用定时器2作为串行端口的发送波特率发生器，用定时器1作为接收波特率发生器
T2CON.3	EXEN2	CBH	定时器2外部控制启用位。置为1时，若T2EX信号发生从1到0的跳变，则进行捕获或重载操作
T2CON.2	TR2	CAH	定时器2运行控制位。由软件置位或清除，置1时定时器启动，清零时定时器停止
T2CON.1	C/ $\overline{T2}$	C9H	定时器2计数器/定时器选择位：1＝计数器，0＝定时器
T2CON.0	CP/ $\overline{RL2C}$	C8H	定时器2捕获/重载标志。当置为1时，如果EXEN2＝1且T2EX信号发生从1到0的跳变，则发生捕获操作，置为0时，如果定时器溢出或EXEN2＝1且T2EX信号发生从1到0的跳变，则发生自动重载操作。如果RCLK或TCLK＝1，那么该位被忽略

104

4.10.1 自动重装载模式

T2CON中的捕获/重载位决定了定时器2工作在自动重载还是捕获模式，当CP/ $\overline{RL2}$ ＝0时，定时器2工作在自动重载模式，TL2/TH2用作定时器寄存器，RCAP2L和RCAP2H中保存重载值。与定时器0和定时器1的重载模式不同，定时器2在自动重载模式中仍然是16位定时器。

当TL2/TH2的数值从FFFFH变成0000H时，发生重载操作，同时将定时器2的溢出标志位TF2置1。TF2的状态可由软件查询，也可以编程使其在被置位时触发中断。无论哪种方式，必须在TF2被再次置1之前用软件清零。

105

另外，如果将T2CON中的EXEN2位设置为1，当引脚T2EX的信号发生从1到0的跳变时，也将发生重载操作，在8052芯片上，T2EX是P1.1的第二功能。T2EX信号若是发生从1到0的跳变，8052会同时将定时器2的标志位EXF2置1。同TF2一样，EXF2的状态既可以用软件查询，也可以编程使其在被置位时触发中断。EXF2必须由软件清除。图4-8描述了定时器2的自动重载模式。

4.10.2 捕获模式

CP/ $\overline{RL2}$ ＝1时，定时器2工作于捕获模式。这时定时器2作为16位定时器，TL2/TH2发生从FFFFH到0000H的跳变时TF2位被置1。TF2的状态可用软件查询，也可以编程使其被置1时触发中断。

要使捕获功能生效，必须将T2CON的EXEN2位置1。如果EXEN2＝1，当T2EX(P1.1)的信号发生从1到0的跳变时，寄存器TL2/TH2中的数值被“捕获”，并送入寄存器RCAP2L和RCAP2H中，同时EXF2标志被置1。EXF2的状态可以用软件

查询，也可以编程使其被置1时触发中断。图4-9描述了定时器2的捕获模式。

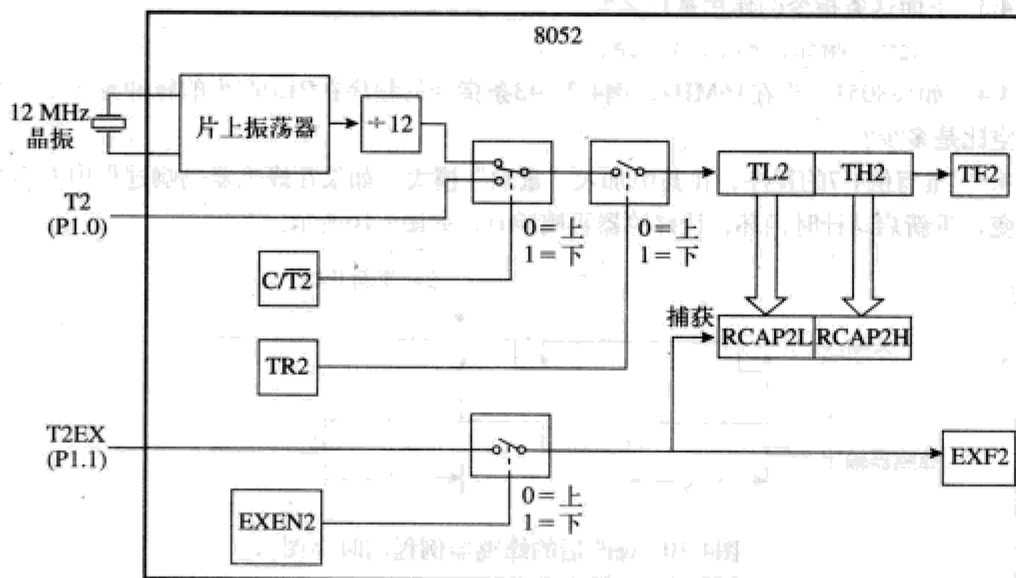


图4-9　工作在16位捕获模式的定时器2

4.11　波特率发生器

定时器的另外一个用途是为片上串行端口提供波特率时钟信号。这个功能在8051上由定时器1提供，在8052上由定时器1和/或定时器2提供。如何产生波特率将在第5章中讨论。

小结

本章介绍了8051和8052微控制器的定时器。本章示例中提供的程序存在一个共性但很有限制性的问题。这些程序一直处在循环中等待定时器溢出，消耗了所有的CPU时间。出于学习的目的这样做没有问题，但是对于实际面向控制的微控制器应用，CPU必须有时间执行其他任务和响应外部事件，如操作者从键盘输入相关参数等。在关于中断的章节中，将介绍如何在“中断驱动”环境中使用定时器。那时不再用软件查询定时器溢出标志，而是由其触发中断。当某个动作引起定时器中断时（也许是翻转某个端口位），另一个程序会暂时中断主程序的运行。通过中断，系统实现了同时执行数个任务。

习题

4.1　编写程序，在引脚P1.5上产生频率为100kHz的方波（提示：不要使用定时器）。

4.2　下面这条指令的作用是什么？

```
SETB   8EH
```

4.3 下面这条指令的作用是什么？

```
MOV   TMOD, #11010101B
```

4.4 如果8051工作在16MHz，例4-2 中3条指令的程序在P1.0产生的脉冲频率是多少？其占空比是多少？

4.5 重写例4-7的程序，在其中加入“重启”模式。如果在蜂鸣器鸣响过程中发生1到0的跳变，重新启动计时循环，让蜂鸣器再鸣响1s，如图4-10所示。

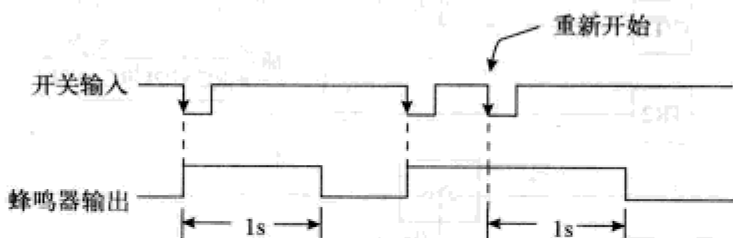


图4-10 修改后的蜂鸣器例程的时序图

4.6 编写程序，利用定时器0在引脚P1.2上产生频率为12kHz的方波。

4.7 设计一个“十字转门”程序，利用定时器1监测何时第10 000个人进入集市。假设：(1) 1个转门传感器连接在引脚T1上，每次转门转动就会产生1个脉冲；(2) P1.7连接1个指示灯，当P1.7 = 1时点亮；P1.7 = 0时熄灭。计算T1端发生“事件”的次数并在第10 000个人进入集市的时候点亮指示灯，如图4-11所示。

107

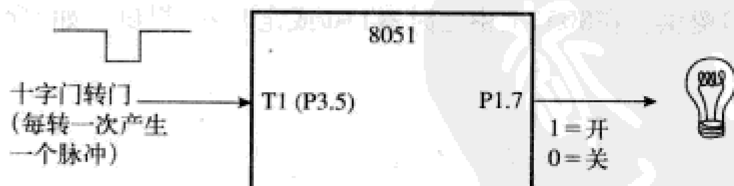


图4-11 十字门转门问题

4.8 国际通行的乐器调音的标准音是“中央C上的A”，频率为440Hz。编写程序，产生440Hz的脉冲信号，并驱动连接在引脚P1.1上的扬声器发出440Hz的音调（如图4-12所示）。由于TL1/TH1中数值的舍入，输出频率会有稍许偏离。实际的输出频率是多少？百分率误差是多少？使用频率为多少的晶振可以使编写的程序产生精确的440Hz脉冲信号？

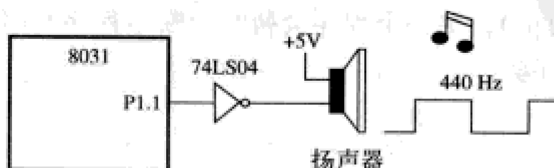


图4-12 扬声器接口

4.9　利用定时器编写程序，在P1.0产生频率为500kHz、占空比为30%的脉冲信号（占空比＝高电平时间÷周期）。

4.10　图4-13所示电路能够为T2提供十分精确的60Hz信号，其中的60Hz余弦信号取自电源变压器的副边线圈的抽头，然后再经本电路整形成60Hz的方波信号。初始化定时器2，使其接受T2的时钟信号，并每秒溢出1次。每次溢出时更新8052内部存储器中的时间（时、分和秒）值。存储器地址为50H（时）、51H（分)和 52H（秒)。有关定时器更多的例程及问题请参阅第6章。

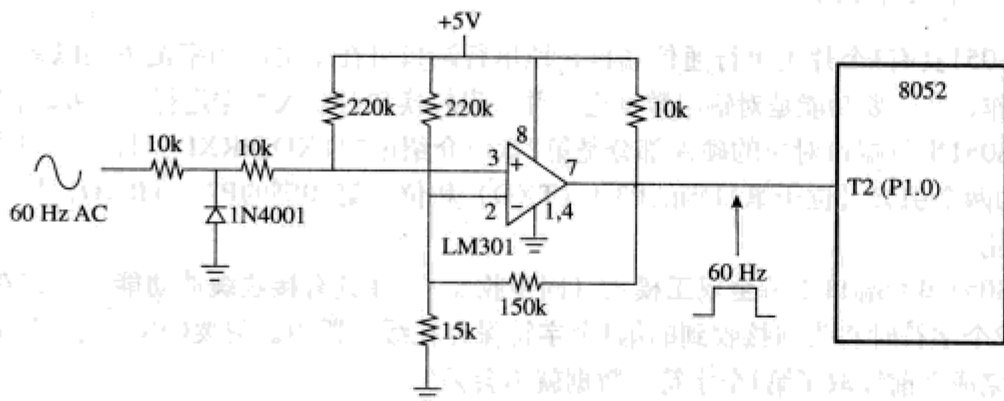


图4-13　60Hz时基

4.11　(a) 编写程序，在P1.0端口产生占空比为30%的矩形脉冲信号，在P2.0产生方波信号。
(b) 由此计算两种信号的舍入误差及由指令执行占用时间造成的误差各是多少。

4.12　编写程序，在P1.0产生占空比为20%的2.5kHz的矩形脉冲信号，要求在每条指令后写出详细注释。

4.13　在将外部晶振由12MHz替换成16MHz的情形下，编写程序，在P1.0产生尽可能高的频率的脉冲信号，并计算这个最高频率值及信号的占空比。

4.14　假设需要利用8051产生1小时的定时间隔，应该选择何种定时器工作模式？为什么？

4.15　假设需要在P1.0产生21kHz、占空比为10%的方波，选择哪种定时器工作模式最为合理？为什么？

4.16　8051定时器的最大计数值是多少？如果将8051定时器作为事件计数器使用，将会产生什么效果？为什么？

108

4.17　你认为16位的定时器都能用来做什么事情？解释原因。

4.18　(a) 编写程序，在P2.0产生3kHz、占空比为30%的脉冲信号，展示整个工作过程，在每条语句之后加详细注释，并做出一些必要的调整来补偿由于指令执行所造成的时间消耗。
(b) 在上面的程序中是否存在截断误差？如果有，计算其百分误差；如果没有，解释原因。

109

第5章 串行端口操作

5.1 本章简介

8051具有1个片上串行通信端口，该串行端口可在很宽的频率范围内以多种模式工作，其主要功能是对输出数据进行并—串转换和对输入数据进行串—并转换。

8051串行端口对应的硬件部分是第2章中介绍过的TXD和RXD引脚。它们是端口3的两个引脚［位于第11脚的P3.1（TXD）和位于第10脚的P3.0（RXD）］的第二功能。

8051串行端口支持**全双工**模式（同步收发），并具有**接收缓冲功能**，允许在接收第2个字符时将先前接收到的第1个字符保存在缓冲器中。只要CPU在第2个字符接收完成之前读取了第1个字符，数据就不会丢失。

串行端口的工作频率（即波特率）可以是固定的（由8051的片上振荡器驱动），也可以是变化的。如果使用可变波特率，波特率时钟信号由定时器1提供，在使用时必须对其做相应的编程。（在8032/8052上，定时器2也可提供波特率时钟）。

两个特殊功能寄存器［串行端口缓冲寄存器（SBUF）和串行端口控制寄存器（SCON）］完成串行端口的软件控制功能。

5.2 串行通信

在开始讨论8051的串行端口操作之前，我们先接触一些串行通信的基本概念。串行通信即只通过一根通信线完成数据位的传输。数据以比特流的形式按同步或异步格式传送。**同步**串行通信按照参考时钟同步传送整个字符数据包，而**异步**串行通信在任意时刻随机地传送一个字符，不依赖于时钟信号。例如，每一个按键信号从键盘传送到计算机就是异步通信模式，因为敲击键盘的速率是不固定的而且发生在随机的时刻。如果与此同时两台计算机在进行同步通信，那么在整个通信期间二者必须被同步于一个相同的参考时钟。

5.3 串行端口缓冲寄存器

串行端口缓冲寄存器（SBUF）的地址是99H，实际是2个缓冲器，写SBUF的操作完成待发送数据的加载，读SBUF的操作可获得已接收到的数据。两个操作分别对应两个不同的寄存器，1个是只写发送寄存器，1个是只读接收寄存器

（见图5-1）。

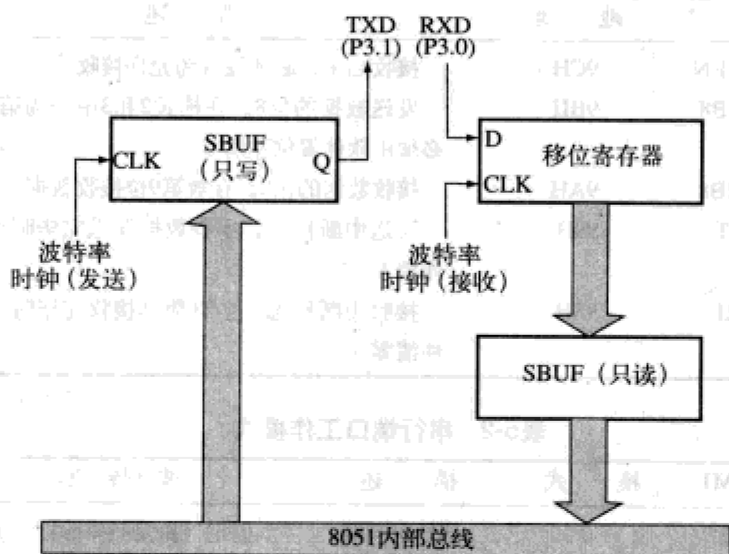


图5-1 串行端口结构图

注意，图5-1中的串入并出移位寄存器在接收数据被传送到接收只读寄存器之前对其进行计时。该移位寄存器是串行端口能提供缓冲功能的重要部件，只有当8位数据全部接收完成后才将其传送到接收只读存储器。这就保证了接收数据进行过程当中先前接收的数据仍可完整地保存在接收只读存储器中。

112

5.4 串行端口控制寄存器

串行端口控制寄存器（SCON）的地址是98H，可以位寻址，包括串行端口的状态位和控制位。状态位用于指示字符接收完成，在软件中被用于测试或者用于编程，以引发中断。同时，写控制位可以设置8051串行端口的工作模式（见表5-1和表5-2）。

使用串行端口之前必须对SCON寄存器做相应的初始化，选择合适的工作模式并进行其他设置。例如，下面的指令

```
MOV    SCON, #01010010B
```

初始化串行端口为模式1（SM0/SM1＝0/1），允许接收（REN＝1），并置位发送中断标志（TI），指示发送器准备就绪。

表5-1 SCON（串行控制寄存器）简表

位	符号	地址	描述
SCON.7	SM0	9FH	串行端口模式位0（见表5-2）
SCON.6	SM1	9EH	串行端口模式位1（见表5-2）
SCON.5	SM2	9DH	串行端口模式位2，允许模式2和3进行多处理器通信控制位；如果接收到的第9位数据为零，那么不激活RI

（续）

位	符号	地址	描述
SCON.4	REN	9CH	接收启用，必须设置为允许接收
SCON.3	TB8	9BH	发送数据的位8。在模式2和3中作为第9位发送数据，必须由软件置位和清零
SCON.2	RB8	9AH	接收数据的位8。存放第9位接收数据
SCON.1	TI	99H	发送中断标志，字符数据发送完毕时硬件置位，由软件清零
SCON.0	RI	98H	接收中断标志，字符数据接收完毕时硬件置位，由软件清零

表5-2 串行端口工作模式

SM0	SM1	模式	描述	波特率
0	0	0	移位寄存器	固定（振荡频率除以12）
0	1	1	8位UART	可变（由定时器设置）
1	0	2	9位UART	固定（振荡频率除以12或除以64）
1	1	3	9位UART	可变（由定时器设置）

5.5 工作模式

8051的串行端口有4种工作模式，通过向SCON中的SM0和SM1位写1或写0进行选择。其中3种模式为异步通信，每个发送和接收的字符都带有1个启始位和1个停止位。熟悉微型计算机RS232C串行端口操作的读者会发现二者有许多相似之处。在第4种模式中，串行端口被作为1个简单的位移寄存器来使用。下面分别对每种模式做简要介绍。

5.5.1 8位移位寄存器（模式0）

置SCON的SM0和SM1为0可以选择串行端口工作模式为模式0，串行端口作为8位移位寄存器用。通过RXD接收或发送数据，TXD输出移位时钟信号。首先用最低有效位（LSB）发送或接收8位数据。

113

波特率固定为片上振荡器频率的1/12。在这个模式下术语RXD和TXD已经不再代表原来的含义。RXD线既用于输入数据也用于输出数据，TXD线用作时钟。

采用任何一条指令将数据写入SBUF即启动发送。数据通过RXD线（P3.0）移出，时钟脉冲通过TXD线（P3.1）输出。每个被发送的“位”将在RXD引脚上维持1个机器周期（有效时间）。在每个机器周期中，时钟信号在S3P1期间变为低电平，并在S6P1期间回到高电平。图5-2是数据输出的时序图。

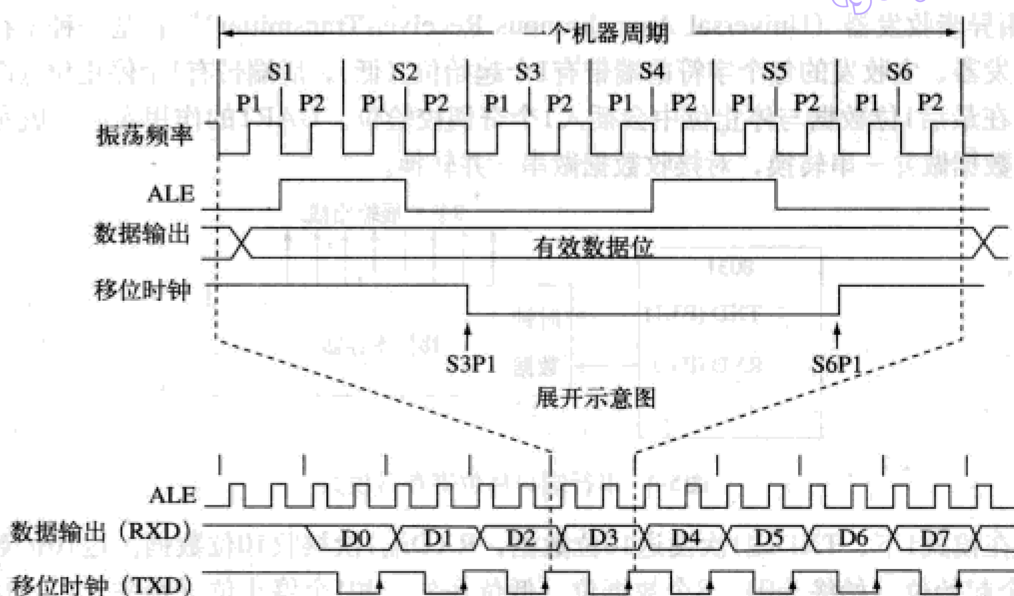


图5-2　串行端口工作于模式0时的发送时序

当接收启用位（REN）被置为1、接收中断标志（RI）被置为0时，接收开始。在实际开发中，通常的做法是在程序初始化串行端口的时候置REN为1，要开始接收数据时清除RI。当RI被清零时，时钟脉冲从TXD线输出，开始下一个机器周期，数据被定时于RXD线。显然，外部附加电路需要根据TXD线的时钟信号同步发送数据。数据的同步输入发生在TXD时钟信号的上升沿（见图5-3）。注意，在这种运行模式中，8051串行端口和附加电路之间的数据传送是通过同步通信进行的（即8051串行端口和附加电路都与TXD上的时钟信号同步）。下文即将介绍，串行端口的其他运行方式都是通过异步通信进行的。

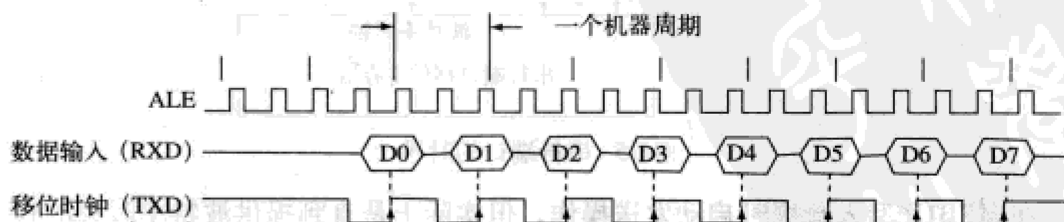


图5-3　串行端口工作于模式0时的接收时序

移位寄存器模式可用于扩展8051的输出能力。把TXD、RXD和1个串－并移位寄存器芯片连接起来，可以扩展出8条输出线（见图5-4）。若要求更多的扩展，可在第1个移位寄存器后级联更多的移位寄存器。

5.5.2　8位可变波特率UART（模式1）

在模式1下，8051的串行端口是1个具有可变波特率的8位UART。UART［即

"通用异步收发器（Universal Asynchronous Receiver/Transmitter）"］是一种串行数据收发器，它收发的每个字符前端带有1个起始位（低），后端带有1个停止位（高）。有时在最后1位数据与停止位中会插入1个奇偶校验位。UART的作用实际上就是对发送数据做并－串转换，对接收数据做串－并转换。

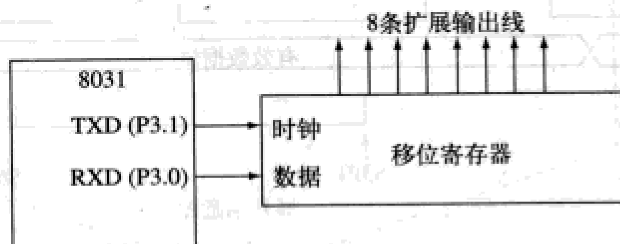


图5-4 串行端口移位寄存器模式

在模式1下，TXD端1次发送10位数据，RXD端1次接收10位数据。这10位数据是1个起始位（始终为0）、8个数据位（低位在先）和1个停止位（始终为1）。接收时，停止位被送入SCON寄存器的RB8。对于8051，波特率取决于定时器1的溢出速率；对于8052，波特率取决于定时器1或定时器2的溢出速率（发送波特率取决于其中之一，接收波特率取决于另一个）。

115

串行端口位移寄存器在模式1、2、3下由1个4位16分频计数器提供时钟和同步信号。计数器的输出信号就是波特率时钟（见图5-5）。计数器的输入信号可通过软件选择，具体情况将在后面讨论。

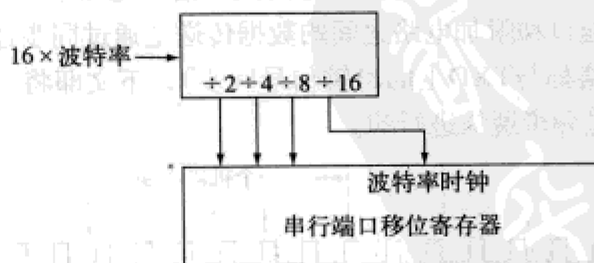


图5-5 串行端口的时钟

在向SBUF写入数据时启动发送操作，但实际上是直到提供波特率时钟的16分频计数器在写动作后的首次翻转时才真正开始发送的。将数据移送到TXD，包括1个起始位、8个数据位、1个停止位。每一位的持续时间是串行端口波特率的倒数。在停止位被传送到TXD之后马上置位发送中断标志位RI（见图5-6）。

当RXD信号发生1到0的跳变时，8051开始接收数据。16分频计数器立即复位并与输入数据流保持同步（每计数达到16的倍数就接收1位数据，如此重复）。对输入数据的采样在16次计数的中间进行。

接收器具有"虚假起始位检测"功能。RXD端口上的信号发生1到0的跳变后，

必须连续8次计数都保持“0”才会被认为是起始位，否则接收器认为该负跳变是噪声信号并将其忽略。如果是噪声信号，接收器重新复位进入空闲状态，等待下一次负跳变的发生。

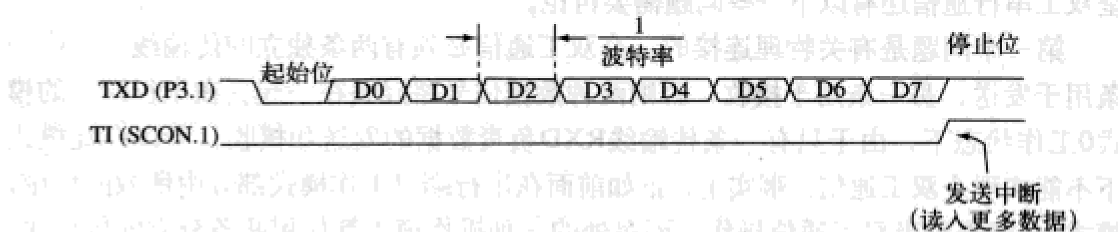


图5-6 串行端口发送中断标志位TI的设置

如果起始位有效，继续接收字符。接收器跳过起始位，将8位数据逐一移入串行端口移位器。一旦8位数据接收完毕，接着进行如下操作：

(1) 第9位（停止位）被送入SCON中的RB8；

(2) 8位数据被载入SBUF；

(3) 接收中断标志（RI）被置位。

但是，上面的动作只有在满足下面的条件下才会发生：

(1) RI＝0；

(2) SM2＝1且接收到的停止位为1，或者SM2＝0。

要求RI＝0是为了保证软件已经读取了前面接收的字符（并清零了RI）。第(2)个条件看上去似乎比较复杂，但仅在多处理器通信模式中（见后文）才会用到。其意义是：在多处理器通信模式下，如果第9位为0则不要置位RI。

5.5.3 9位固定波特率UART（模式2）

当SM＝1且SM0＝0时，串行端口工作在模式2下，成为1个9位的固定波特率UART。每次发送或接收11位数据，包括1个起始位、8个数据位、1个可编程位（第9数据位）和1个停止位。在发送时，第9数据位是TB8中的内容（可能是奇偶校验位）。在接收时，第9数据位被送入RB8中。模式2下的波特率是片上振荡器频率的1/32或1/64（见5.9节）。

5.5.4 9位可变波特率UART（模式3）

在模式3下，串行端口是1个9位可变波特率UART，工作方式与模式2基本相同，唯一的差别是其波特率是可变的，由定时器提供。事实上，模式1、模式2和模式3非常相似。差别在于波特率（模式2固定，模式1、模式3可变）和数据位的长度（模式1为8位，模式2、模式3为9位）。

5.6 全双工串行通信讨论

8051的串行端口支持全双工串行通信，即允许发送和接收数据同时进行。关于全双工串行通信还有以下一些问题需要讨论。

第一个问题是有关物理连接的。全双工通信必须有两条独立的传输线，其中一条用于发送，另一条用于接收，否则两种数据信号将混叠在一起。在串行端口的模式0工作状态下，由于只有一条传输线RXD负责数据的发送和接收，所以在此模式下不能实现全双工通信。事实上，正如前面在串行端口工作模式部分中所做的讨论，模式0只能提供半双工通信操作。而另外的三种操作模式都使用两条分立的传输线，TXD和RXD分别用来发送和接收，所以都支持全双工通信。

第二，必须考虑全双工串行通信的时序和同步。当学生知道串行端口可以进行全双工通信之后，一个共性问题就会被提出来："假设串行端口正在向某一串行设备发送数据，那么此时8051和外部串行设备是处于同步状态的。如果此时在RXD线上探测到接收数据，将会发生什么？"这引出了一个有趣的问题。为了解答该问题需要了解串行端口的内部结构，如图5-1所示，8051包括两个在物理上彼此独立的SBUF寄存器。发送只写寄存器同步于发送波特率时钟，接收只读寄存器同步于接收波特率时钟。如前面章节的讨论（见5.5.2节的模式1）一样，这两个波特率时钟都基于定时器1的溢出率。尽管如此，实际上两个波特率时钟还是彼此独立和不相关的。

117

发送操作在向SBUF写入数据时被启动，但确切的开始是在探测到下一个发送波特率时钟脉冲的时候。同时，如果有任何接收数据被探测到，标志起始位的1到0的电平跳变立即复位接收波特率时钟，所以此时8051实质上是同步于与其相连接的外部串行设备的。

在本章的结尾部分将给出一个串行端口全双工通信的范例。

5.7 串行端口寄存器的初始化和访问

5.7.1 接收启用

要接收数据，必须用软件设置SCON寄存器中的接收启用位（REN）。通常这个工作在程序开始阶段初始化串行端口、定时器等设备的时候完成。置位REN有两种方法，一种是用指令

```
SETB  REN
```

来明确地设置REN，或者用指令

```
MOV  SCON, #xxx1xxxxB
```

设置REN，同时根据需要修改SCON的其他位（为设置串行端口的工作模式，位x必

须置为1或0)。

5.7.2 第9数据位

在模式2和模式3中，待发送的第9位数据必须由软件写入TB8。接收到的第9位数据被送入RB8中。在软件中第9位数据是否有用取决于串行端口通信设备的特征(在多处理器通信模式中第9数据位起着重要的作用，见后文)。

5.7.3 加入奇偶校验位

第9数据位常见的用途是作为字符的奇偶校验位。在第2章中已经讨论过，程序状态字（PSW）中的*P*位是奇偶标志，用来对累加器*A*中内容进行偶数奇偶校验，每过一个机器周期，其内容随累加器*A*中的数值变化1次。如果通信中需要在8位数据后加上偶数奇偶校验位，则可以用下面的指令发送累加器*A*中的8位数据并以第9数据位作为奇偶校验位：

```
MOV   C,P              ; PUT EVEN PARITY BIT IN TB8
MOV   TB8,C            ; THIS BECOMES THE 9TH DATA BIT
MOV   SBUF,A           ; MOVE 8 BITS FROM ACC TO SBUF
```

如果需要做奇数奇偶校验，指令可修改为：

```
MOV   C,P              ; PUT EVEN PARITY BIT IN C FLAG
CPL   C                ; CONVERT TO ODD PARITY
MOV   TB8,C
MOV   SBUF,A
```

118

当然，奇偶校验不仅限于在模式2和模式3中使用。在模式1下，发送的8位数据可由7位数据和1位奇偶校验位组成。下面的指令可用于发送1个7位ASCII代码和1个奇偶校验位：

```
CLR    ACC.7           ; ENSURE MSB IS CLEAR
                       ; EVEN PARITY IS IN P
MOV    C,P             ; COPY TO C
MOV    ACC.7,C         ; PUT EVEN PARITY INTO MSB
MOV    SBUF,A          ; SEND CHARACTER
                       ; 7 DATA BITS PLUS EVEN PARITY
```

5.7.4 中断标志

SCON寄存器中的接收和发送中断标志（RI和TI）在8051的串行端口通信中起着重要作用。这两个标志位都是由硬件置位，且必须由软件清零。

在典型的应用中，RI在字符接收完成的时候被置1，指示“接收缓冲区已满”。可以用软件检查RI的状态，或者编程在RI被置1时触发中断（中断在第6章中讨论）。如果软件要从与串行端口相连的设备（也许是1台视频显示终端）上读取1个字符，那么必须等待RI被置1，然后清除RI并从SBUF中读取字符。指令如下：

```
WAIT:     JNB     RI,WAIT      ; CHECK RI UNTIL SET
          CLR     RI           ; CLEAR RI
          MOV     A,SBUF       ; READ CHARACTER
```

TI在字符发送完成的时候被置1，指示“发送缓冲区已空”。如果软件要通过串行端口发送字符，首先必须检查串行端口是否就绪。换句话说，必须等待前面的字符发送完毕才能发送后面的字符。下面的指令可发送累加器中的字符：

```
WAIT:   JNB   TI,WAIT    ;CHECK TI UNTIL SET
        CLR   TI         ;CLEAR TI
        MOV   SBUF,A     ;SEND CHARACTER
```

上面的接收和发送指令通常是标准字符输入和输出子例程的一部分。在例5-2和例5-3中，对此有更详细的描述。

5.8 多处理器通信

8051串行端口的模式2和模式3专门用作多处理器通信。在这两个模式中，接收的是9位数据，第9位进入RB8。可以通过程序设置，使得串行端口接收到停止位后，只有当RB8＝1时，串行端口中断才会被触发。要启用这个功能，需置SCON寄存器中的SM2为1。如图5-7所示，在使用多个8051微控制器的场合中，可以利用这一特点构成主从网络环境。

119

当主处理器准备向几个从处理器中的某一个发送1个数据块时，首先发送1个地址字节，以辨认目标处理器。地址字节与数据字节的差别在第9位数据位，地址字节的第9数据位为1，数据字节的第9数据位为0。一个地址字节会中断所有从处理器，所有从处理器都检查接收到的字节，以判断自己是不是目标从处理器。被寻址的从处理器清除它的SM2，并准备接收即将到来的数据字节，未被寻址的从处理器则保持它们的SM2为1，忽略接收到的数据字节，继续它们自己的任务，直到一个新的地址字节来临。可以设计出特殊的结构，使得主处理器向从处理器发送数据的同时，从处理器也可以向主处理器发送数据。关键是在建立了主从连接之后不要再使用第9数据位（否则会错误地选择其他从处理器）。

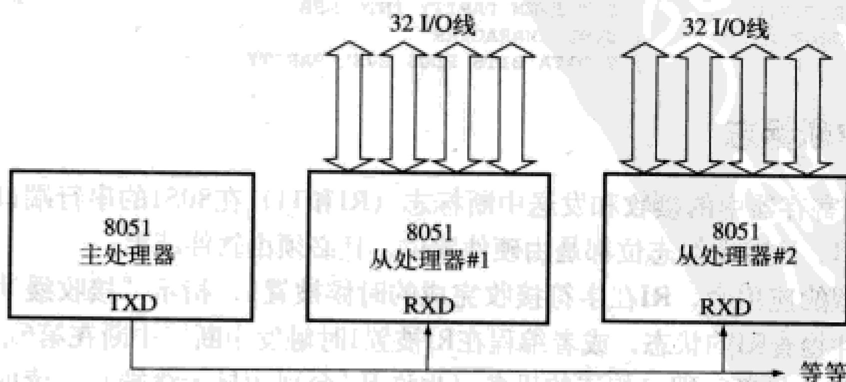


图5-7 多处理器通信

在模式0中SM2无效，在模式1中可用于检查停止位的有效性。在模式1下接收数据，如果SM2＝1，只有接收到有效的停止位后，接收中断才能发生。

5.9 串行端口波特率

如表5-2中所列，模式0和模式3的波特率是固定的，模式0的波特率为片上振荡器频率的1/12。通常8051的片上振荡器由晶振驱动，但是同样也可以使用另一种时钟信号源（参见第2章）。如果标准振荡器频率为12MHz，模式0的波特率就是1MHz（见图5-8a）。

系统复位后，模式2的默认波特率是振荡器频率的1/64。波特率还受到电源控制寄存器PCON中的第7位的影响。PCON寄存器的第7位是SMOD（波特率倍增）位。SMOD被置1时，模式1、模式2和模式3下的波特率会变成原来的2倍。在模式2下，波特率可以从1/64振荡器频率的默认值（SMOD＝0）变为振荡器频率的1/32（SMOD＝1）（见图5-8b）。

120

由于PCON不可位寻址，因此，若要设置SMOD同时又不改变其他位，就需要以“读—改—写”的方式操作。下面的指令可以设置SMOD：

```
MOV     A,PCON          ;GET CURRENT VALUE OF PCON
SETB    ACC.7           ;SET BIT 7 (SMOD)
MOV     PCON,A          ;WRITE VALUE BACK TO PCON
```

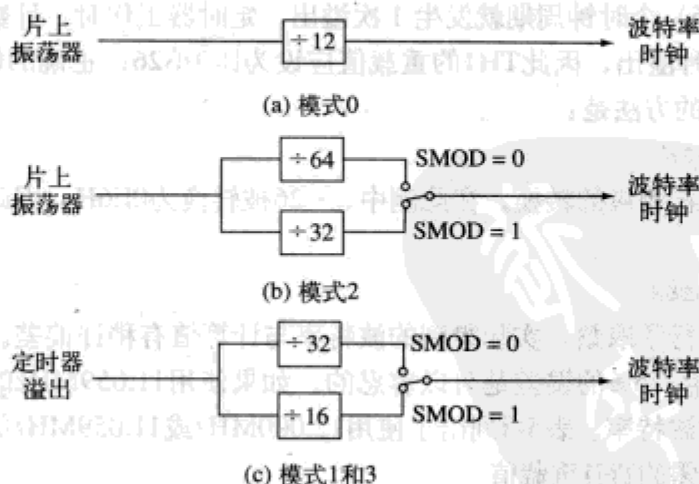


图5-8 串行端口的时钟源

8051在模式1和模式3下的波特率由定时器1的溢出速率决定。由于定时器工作在相对较高的频率，因此其溢出速率还要除以32（如果SMOD＝1，则除以16）才作为串行端口波特率时钟，如图5-8c所示。8052在模式1和3下的波特率由定时器1或定时器2的溢出率决定，也可由它们共同决定。

定时器1作为波特率发生器

考虑仅1片8051的情况，通常的波特率发生方法是将TMOD初始化为8位自动重装载模式（定时器1的模式2），并将合适的重载值送入TH1，以产生满足波特率需

要的正确的溢出率。TMOD用下面的指令初始化：

```
MOV    TMOD, #0010xxxxB
```

121 其中x代表1或0，根据定时器0的需要选择。

上面的方法并不是唯一的方法。非常低的波特率可以利用定时器1的16位模式实现，即模式2，并设置寄存器TMOD＝0001xxxxB。这种情况下，每次溢出后TH1/TL1寄存器必须用软件重新装载，可利用一段中断服务子例程完成。另一种选择是利用T1（P3.5）的外部信号作为定时器1的时钟信号。不管使用什么方法，波特率始终是定时器1溢出速率的1/32（如果SMOD＝1，则为1/16）。

模式1和模式3的波特率由下式确定：

波特率＝定时器1溢出率÷32

例如，波特率为1200，定时器1的溢出速率可由下式计算：

1200＝定时器1溢出率÷32

定时器1溢出率＝38.4kHz

如果片上振荡器由12MHz的晶振驱动，那么定时器1的时钟频率是1MHz，或表示为1000kHz。由于定时器的溢出速率必须为38.4kHz，因此，必须每1000÷38.4＝26.04（取整为26）个时钟周期就发生1次溢出。定时器工作时，计数依次增加，在从FFH变到00H时溢出，因此TH1的重载值应设为比0小26，正确的值是−26。装载TH1重载最简单的方法是：

```
MOV  TH1,#-26
```

编译器会进行相应的转换。在此例中，−26被转换为0E6H，因此上面的指令等同于

```
MOV  TH1,#0E6H
```

由于前面进行了取整，实际得到的波特率与计算值有稍许偏差。通常，在异步（启／止）通信中，5%的误差是可以容忍的。如果使用11.059MHz的晶振，则可以得到完全准确的波特率。表5-3列出了使用12.000MHz或11.059MHz的晶振时，产生

122 各常用波特率所需的TH1重载值。

表5-3　波特率简表

波特率	晶振频率	SMOD	TH1重载值	实际波特率	误差
9 600	12.000 MHz	1	−7(F9H)	8 923	7%
2 400	12.000 MHz	0	−13(F3H)	2 404	0.16%
1 200	12.000 MHz	0	−26(E6H)	1 202	0.16%
19 200	11.059 MHz	1	−3(FDH)	19 200	0
9 600	11.059 MHz	0	−3(FDH)	9 600	0
2 400	11.059 MHz	0	−12(F4H)	2 400	0
1 200	11.059 MHz	0	−24(E8H)	1 200	0

例5-1 初始化串行端口

编写一段指令，初始化串行端口。使其以波特率2400工作在8位UART模式，以定时器1作为波特率发生器。

解答：对于此例，必须初始化4个寄存器：SMOD、TMOD、TCON和TH1。各寄存器所需设置值如下：

	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
SCON:	0	1	0	1	0	0	1	0
	GTE	C/T	M1	M0	GTE	C/T	M1	M0
TMOD:	0	0	1	0	0	0	0	0
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
TCON:	0	1	0	0	0	0	0	0
TH1:	1	1	1	1	0	0	1	1

置SM0/SM1＝0/1使得串行端口工作在8位UART模式。REN位置1允许串行端口接收字符。置TI＝1指示发送缓冲区为空，可以发送字符。对于TMOD，置M1/M0＝1/0使定时器1工作在8位自动重装载模式。置TCON寄存器的TR1＝1启动定时器1。由于其余控制位或模式位在本例中没有用到，所以把它们全都设置为0。

讨论：TH1的重装载值必须使其溢出速率为2400 × 32＝76.8kHz。假设8051工作于12MHz晶振频率下，则定时器1的时钟频率为1MHz（1000kHz）。每次溢出前的时钟脉冲个数为1000/76.8＝13.02（取整为13）。因此重载值为−13或0F3H。

初始化指令如下。

例5-1的答案 8051串行端口例子（串行端口的初始化）

```
8100            5           ORG   8100H
8100  759852    6    INIT:  MOV   SCON,#52H  ;serial port,mode 1
8103  758920    7           MOV   TMOD,#20H  ;timer 1, mode 2
8106  758DF3    8           MOV   TH1,#-13   ;reload count for 2400 baud
8109  D28E      9           SETB  TR1        ;start timer 1
                10          END
```

例5-2 输出字符子例程

编写一段子例程，命名为OUTCHR，将8051累加器*A*中的7位ASCII字符代码通过串行端口发送出去，同时以第8位作为奇数奇偶校验码。要求从子例程返回后，累加器*A*的数值与调用子例程前相同。

解答：本例以及例5-3解释了在带有RS232终端的微型计算机系统中最常用的两个子例程，字符输出（OUTCHR）和字符输入（INCHR）。

```
8100            5              ORG   8100H
8100  A2D0      6    OUTCHR:   MOV   C,P        ;put parity bit in C flag
8102  B3        7              CPL   C          ;change to odd parity
8103  92E7      8              MOV   ACC.7,C    ;add to character code
8105  3099FD    9    AGAIN:    JNB   TI,AGAIN   ;Tx empty? no:check again
8108  C299      10             CLR   TI         ;yes: clear flag and
810A  F599      11             MOV   SBUF,A     ;send character
```

```
810C   C2E7    12          CLR   ACC.7      ;strip off parity bit and
810E   22      13          RET              ;  return
               14          END
```

讨论：前3条指令的功能是将奇数奇偶校验码送入累加器的位7。由于PSW中的*P*位是累加器内容的偶数奇偶校验，因此在被送入ACC.7之前先要取反。JNB指令是1个等待循环，不断检测发送中断标志（TI）的状态直到其为1。当TI被置1后（这时前1个字符发送过程已经完成），TI被清除，接着累加器*A*中的字符被写入串行端口缓冲区（SBUF）。数据在为串行端口提供时钟信号的16分频计数器下一次计数满翻转的时候开始发送（见图5-5）。最后，ACC.7被清除，以使返回的数值与传递给子例程的7位代码相同。

OUTCHR子例程是系统程序的一小部分，本身起不到太大作用。更高层的程序调用该子例程发送单独的1个字符或是1个字符串。例如，下面的指令向与8051串行端口连接的串行设备发送字符Z的ASCII代码：

```
MOV A,#'Z'
CALL OUTCHR
(continue)
```

本章最后的习题5.1中，进一步扩展了这个思路，利用OUTCHR子例程构成1个OUTSTR（字符串输出）子例程，用于向与串行端口连接的串行设备发送ASCII字符代码序列（以空字符00H结束）。

例5-3　输入字符子例程

编写一段子例程，命名为INCHAR，从8051的串行端口获得1个输入字符，并在将其7位ASCII代码送入累加器后返回，默认接收到的字符的第8位为奇数奇偶校验码。如果出现奇偶校验错误，设置进位标志为1。

124

解答：

```
8100           5            ORG   8100H
8100   3098FD  6   INCHAR:  JNB   RI,$       ;wait for character
8103   C298    7            CLR   RI         ;clear flag
8105   E599    8            MOV   A,SBUF     ;read char into A
8107   A2D0    9            MOV   C,P        ;for odd parity in A,
                                             ;P should be set
8109   B3      11           CPL   C          ;complementing correctly
               12                            ;indicates if "error"
810A   C2E7    13           CLR   ACC.7      ;strip off parity
810C   22      14           RET
               15           END
```

讨论：该子例程以循环开始，等待接收中断标志（RI）被置为1，这标志着字符已经被接收到SBUF中，等待读取。当RI被置为1时，JNB指令结束，执行下一条指令。RI被清除，SUBF中的字符代码被读入累加器，PSW中的*P*位是累加器内容的偶数奇偶校验位，因此，如果累加器中第7位的奇数奇偶校验码正确无误的话，*P*位应为1。将*P*位移入进位标志位并取反，如果没有错误，那么CY＝0；如果累加器中有奇偶

校验错误，那么CY=1，通知系统发生了奇偶校验错误。最后，ACC.7被清除，只有7位的ASCII代码被返回给主程序。

例5-4 全双工操作

编写一段程序，实现连续发送存放在发送缓冲器（位于内部RAM的30H到4FH）中的字符。如果在串行端口接收到了字符数据，那么将其存放到位于内部RAM 50H的接收缓冲器内。假设8051串行端口已经被初始化在模式1。

解答：该例子示范了串行端口如何工作在全双工方式，也就是同时完成发送和接收。

```
8100              1          ORG 8100H
8100 7830         2          MOV R0, #30H          ;pointer for tx buffer
8102 7950         3          MOV R1, #50H          ;pointer for rx buffer
8104 209819       4   LOOP:  JB RI, RECEIVE        ;character received?
                  5                                ; yes: process it
8107 209902       6          JB TI, TX             ;previous character
                  7                                ; transmitted?
                  8                                ; yes: process it
810A 80F8         9          SJMP LOOP             ; no: continue checking
810C E6           10  TX:    MOV A, @R0            ;get character from tx
                  11                               ; buffer
810D A2D0         12         MOV C, P              ;put parity bit in C
810F B3           13         CPL C                 ;change to odd parity
8110 92E7         14         MOV ACC.7, C          ;add to character code
8112 C299         15         CLR TI                ;clear transmit flag
8114 F599         16         MOV SBUF, A           ;send character
8116 C2E7         17         CLR ACC.7             ;strip off parity bit
8118 08           18         INC R0                ;point to next
                  19                               ; character in buffer
8119 B850E8       20         CJNE R0, #50H, LOOP   ;end of buffer?
                  21                               ; no: continue
811C 7830         22         MOV R0, #30H          ; yes: recycle
811E 80E4         23         SJMP LOOP             ;continue checking
                  24
8120 C298         25  RX:    CLR RI                ;clear receive flag
8122 E599         26         MOV A, SBUF           ;read character into A
8124 A2D0         27         MOV C, P              ;for odd parity in A,
                  28                               ; P should be set
8126 B3           29         CPL C                 ;complementing
                  30                               ; correctly indicates
                  31                               ; "error"
8127 C2E7         32         CLR ACC.7             ;strip off parity
8129 F7           33         MOV @R1, A            ;store received
                  34                               ; character in buffer
812A 09           35         INC R1                ;point to next location
                  36                               ; in buffer
812B 80D7         37         SJMP LOOP             ;continue checking
                  38         END
```


讨论：该程序首先使用两个寄存器$R0$和$R1$分别指向发送和接收缓冲器，然后检查是否有字符被接收到或者前一个待发送的字符是否已经发送完成。注意，接收操作被优先处理。这是因为接收事件更为紧急，对接收字符来说，8051要依赖于外部设备，

要求接收到的数据必须尽可能快地进行处理和保存，否则可能被后续传来的字符数据覆盖掉。还要回想到串行端口正在第二个字符时还能够最多保持一个字符的数据。如果输入字符被完全接收后，其将被从SBUF读出，并且在将其存放到寄存器*R*1所指定的接收缓冲器的空闲位置之前做奇偶校验。*R*1中存放的地址数据加1指向下一个接收缓冲器的有效位置，并且使程序返回到循环检查状态。当前一个发送事件完成时，程序将首先从发送缓冲器获得待发送字符。*R*0被用于指向发送缓冲器中的下一个字符。在将待发送代码传送到SBUF之前，奇数奇偶检验位被存放到待传送代码的位7。*R*1指向下一个待发送字符的位置，程序也监测发送缓冲器的数据是否已经发送完，如果是则重新开始从头发送，周而复始。最后，程序返回到检查是否有接收或发送事件发生的循环等待状态。 126

例5-5 舍入误差的补偿

计算例5-1中串行端口波特率的舍入误差。假设定时器的重载值设置为－13或0F3H，并计算能够产生精确2400波特率的晶振频率。

解答：该波特率的舍入误差为：

$$\text{舍入误差} = \frac{|\text{波特率}_{desired} - \text{波特率}_{rounded\text{-}off}|}{\text{波特率}_{desired}} \times 100\%$$

$$= \frac{|2400 - 2403.8|}{2400} \times 100\%$$

$$= 0.158\%$$

$$f_{desired} = \frac{\text{波特率}_{desired}}{\text{波特率}_{rounded\text{-}off}} \times f_{rounded\text{-}off}$$

$$= \frac{2400}{2403.8} \times 12\text{MHz}$$

$$= 11.98\text{MHz}$$

讨论：本题中期望波特率是2400，对应于13μs的定时间隔定时器1的重载值为－13，产生的溢出率为76.9kHz，相应的波特率为76.9kHz/32＝2403.8。

在计算同精确的波特率2400bps所对应期望晶振频率的过程中，如本例所示，我们采用了比例方法。下面来验算该期望频率能否产生精确的2400波特率。一个11.98MHz的晶振意味着定时器1的递增率为11.98MHz/12＝0.998MHz，所以其一个计时周期为1/0.998MHz＝1.002μs。而定时器1的重载值为－13，可以计算其溢出周期为13×1.002μs＝1.0326μs，相应的溢出率为1/1.3026μs＝76.77kHz。由此产生的波特率是76.77kHz/32＝2399 baud≈2400 baud。

同11.98MHz相比，11.059MHz的晶振更为常用。在这种情况下，定时器1的重载值必须从−13修正为−12。同样来验算一下，看这样能否产生精确的2400波特率，定时器1的递增率变成11.0592MHz/12＝0.9216MHz，所以其计数周期为1/0.9216MHz＝1.085μs。定时器1的重载值为−12，所以其溢出周期为12×1.085μs＝13.02μs，因此定时器1的溢出率变为1/13.02μs＝76.8kHz。最终产生的波特率为76.8kHz/32＝2400 baud。

有一点是显而易见的，即用晶振频率来消除波特率的舍入误差时，其结果将随定时器1的重载值的不同而变化。

小结

本章介绍了8051串行端口编程的主要内容。本章和上一章中都提及了中断的使用，事实上，关于8051定时器和串行端口的高层次应用通常都需要利用中断来保持输入/输出操作的同步，这些内容是下一章的主题。

习题

下面各题都是关于微型计算机与串行设备接口的典型程序。默认8051的串行端口初始化8位UART模式，波特率时钟信号由定时器1提供。

5.1　编写名称为OUTSTR的子例程，向与串行端口连接的设备（如VDT）发送一段以空字节结束的字符串ASCII代码。假设字符串的ASCII代码存储在外部代码存储器中，主程序在调用该子例程前将字符串地址放入数据指针中。以空字节结束的字符串是指一段以字节00H作为结尾的ASCII代码。

5.2　编写名称为INLINE的子例程，从与串行端口连接的设备接收1行ASCII代码并送入内部数据存储器，存放单元的起始地址为50H。假设ASCII代码行以回车符结束。将回车符与其他代码一起放入上述缓冲区中，并以一个空字节（00H）标志缓冲区的结束。

5.3　编写一段程序，连续向串行端口连接的设备发送字母表（小写）。请利用前面编写的OUTCHR子例程。

5.4　假设OUTCHR子例程可用，编写一段程序，连续向与串行端口连接的设备发送可显示的ASCII字符（从代码20H到7EH）。

5.5　修改上面的程序，允许从键盘输入XOFF和XON代码来暂停和恢复字符输出，忽略其他输入（注：XOFF＝CONTROL-S＝13H，XON＝CONTROL-Q＝11H）。

5.6　假设INCHAR和OUTCHR子例程是现成的，编写一段程序，接收键盘输入的字符并在屏幕上显现出来，同时将小写字符转换为大写字符。

5.7　假设INCHAR和OUTCHR子例程是现成的，编写一段程序，接收与串行端口连接的设备输入的字符并把该字符回显，以句号（.）代替所有控制字符（ASCII代码从00H到1FH，包括7FH）。

5.8　假设OUTCHR子例程是现成的，编写一段程序，清除与串行端口连接的VDT的屏幕，并在屏幕上显示10次你的名字，每次占据1行。大多数VDT的清除屏幕功能键是CONTROL-Z，但在支持ANSI（American National Standards Institute，美国国家标准化组织）标准的VDT上是<ESC>。请在程序中同时使用上面两种方法。

5.9　图5-4介绍了一种扩展8051输出能力的手段。假设已如图5-4所示的配置，编写一段程序，初始化串行端口为移位寄存器模式，并将内部存储器中地址20H处的内容输出到扩展的输出端口上，要求每秒刷新10次。

128

5.10　8051串行端口支持哪种串行通信？半双工还是全双工？为什么？说明串行端口内部元件是如何实现此种串行通信的。

5.11　请说明使用串行端口来进行8051的I/O口扩展能力。该扩展方法的缺点是什么？

5.12　编写名称为OTUCHR9的子例程，通过8051串行端口发送9位编码（$C_8C_7C_6C_5C_4C_3C_2C_1C_0$）。注意：$C_8$存放在$B$寄存器的LSB中，而$C_7C_6C_5\ C_4C_3C_2C_1C_0$存放在累加器中。要求从子例程返回时将所有的寄存器恢复初始状态。

5.13　编写名称为INCHR8的子例程，通过8051串行端口输入8位的扩展ASCII码，并且将该8位代码存放到累加器中后返回。要求第9位为奇偶校验位，而且当出现错误时置位进位标志。

5.14　编写名称为OUTCHR8的子例程，通过8051串行端口发送8位的扩展ASCII码，第9位为奇偶校验位，返回累加器的初始状态值。

5.15　(a) 写一段代码来初始化串行端口，使其以9600波特率工作于9位UART方式，使用定时器1作为波特率时钟，假设晶振频率$f_{osc}=12\text{MHz}$。

(b) 上述初始化所产生的波特率是精确的9600吗？其百分比误差是多少？由此计算能够产生精确9600波特率的晶振频率值。

129

第6章 中　断

6.1 引言

中断是当某条件出现（相应的事件发生）时打断主程序的运行，使CPU转而去执行另外子例程的现象。中断在微控制器应用领域的设计和执行过程中扮演着非常重要的角色。通过中断方式允许系统在执行主程序时可以响应并处理其他任务。**中断驱动系统**给人造成一种微控制器可以同时执行多任务的假象。当然，CPU不能同时执行1条以上的指令，但是它可以暂时停止执行主程序转去执行其他程序，完成后再返回继续执行主程序。从这个角度看，中断响应非常类似于子例程的调用过程。二者的主要区别在于中断响应是由中断驱动系统发起的，而不是像子例程调用那样在主程序流程中预先设定的，中断是系统响应一些和主程序异步的事件，这些事件何时将主程序中断是事先未知的。

在中断响应过程中，系统转去执行的程序被称为**中断服务程序**（ISR）或**中断处理程序**。中断服务程序负责响应中断并执行针对外设的输入输出操作。当中断发生时，主程序暂停执行，系统转到中断服务程序，执行响应的操作，中断服务程序总是以1条“中断返回”指令结束，从中断返回后系统将从断点开始继续执行刚才被打断的主程序。一般来讲，称主程序工作在“**基本级**”而中断服务程序工作在“**中断级**”，有时也用“**前台**”（对应“基本级”）和“**后台**”（对应“中断级”）两个术语。图6-1简要描述了中断的响应时序，其中，（a）表示没有中断的情形，（b）表示在

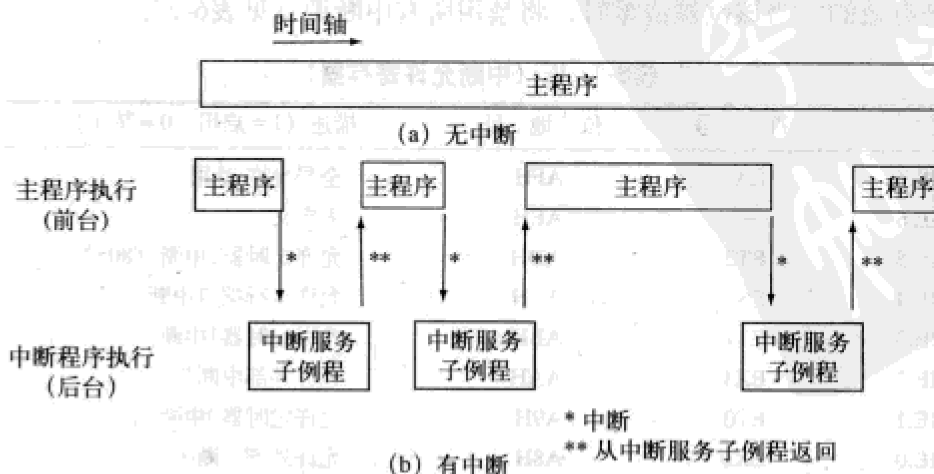


图6-1　在有中断和无中断情况下程序的执行时序

131 "基本级"执行主程序，当中断产生时在"中断级"执行中断服务程序。

使用键盘手动向系统输入信息是中断的典型例子，比如微波炉的操作，主程序在前台控制功率单元实现加热功能。但是，在加热过程中，系统必须能响应用户在微波炉前面板键盘区的输入信息，例如需要缩短或者延长加热时间。当用户按下1个键时随即产生1个中断（或许是信号由高电平变到低电平），主程序被打断。中断服务子例程在后台读取键值然后对加热条件作响应的调整，之后返回到主程序，从端点处继续执行。在本例中关键一点是用户的手动输入操作是随机发生的，也就是说，它的发生是事先未知的和软件不可控的，这就是1个中断。

6.2 8051的中断结构

8051有5个中断源：2个外部中断、2个定时器中断和1个串口中断。8052增加了1个中断源——外部定时器中断。所有中断在系统复位后都是禁用状态，使用时需要分别启用。

当两个以上的中断同时发生或者1个中断发生在CPU正在处理其他中断的期间内，8051通过查询顺序和高低优先级来解决此问题。中断源的查询顺序是固定的，但其优先级是可编程的。

下面开始讨论中断的启用和禁用。

6.2.1 启用和禁用中断

通过可位寻址的特殊功能寄存器IE（中断允许寄存器，地址A8H），每个中断源均可独立地进行启用和禁用的设置。IE中除了每个中断源的独立控制位外，还包括1个全局中断控制位EA，当该位被置1时，即允许全局中断，之后再置位各中断源位才是有效的；当该位被清零时，将禁用所有中断源（见表6-1）。 132

表6-1 IE（中断允许寄存器）

位	符 号	位 地 址	描述（1=启用，0=禁用）
IE.7	EA	AFH	全局允许/禁用
IE.6	—	AEH	未定义
IE.5	ET2	ADH	允许定时器2中断（8052）
IE.4	ES	ACH	允许串行端口中断
IE.3	ET1	ABH	允许定时器1中断
IE.2	EX1	AAH	允许外部中断1
IE.1	ET0	A9H	允许定时器0中断
IE.0	EX0	A8H	允许外部中断0

要想启用1个中断必须置位2个控制位，即各自的独立允许位和全局允许位。例

如，如下指令可以允许定时器1中断：

```
SETB    ET1         ;ENABLE Timer 1 INTERRUPT
SETB    EA          ;SET GLOBAL ENABLE BIT
```

这2条指令也可使用下面的指令代替：

```
MOV  IE, #10001000B
```

尽管这两种方案在系统复位后的执行效果是相同的，但当它们位于某程序中时执行效果会有所差异。很明显，第1种方法不会改变IE寄存器的其余5个控制位，而第2种方法在置位相应控制位的同时将其他控制位清零，所以说，在程序开始执行之初（如上电或复位）采用字节传送指令初始化IE的方法是可以的，但在程序执行过程中需要“实时”启用或禁用中断时，应该采用“置位”和“清零”指令，以免影响IE寄存器的其他控制位。

6.2.2 中断优先级

通过可位寻址的特殊功能寄存器IP（地址0B8H），每个中断源都可独立地被编程设置为高或低优先级（见表6-2）。

表6-2 IP（中断优先级寄存器）

位	符 号	位 地 址	描述（1=高优先级，0=低优先级）
IP.7	—	—	未定义
IP.6	—	—	未定义
IP.5	PT2	0BDH	(8052)定时器2中断（8052）
IP.4	PS	0BCH	串行端口中断的优先级
IP.3	PT1	0BBH	定时器1中断的优先级
IP.2	PX1	0BAH	外部中断1的优先级
IP.1	PT0	0B9H	定时器0中断的优先级
IP.0	PX0	0B8H	外部中断0的优先级

系统复位之后IP寄存器在默认条件下是处于清零状态的，也就是所有中断源都被设置为低优先级。优先级的设置允许1个正在被处理的中断能够被其他中断打断，前提条件是后者的优先级要高于前者。由于8051只有高、低两个优先级，所以，如果系统正在执行1个低优先级的中断服务程序时，高优先级的中断产生，则该中断服务程序将会被打断。反之，高优先级的中断服务程序则不会被打断。

当然，系统运行在“基本级”且没有响应任何中断时，则总可以被某个中断打断而与中断的优先级无关。如果两个不同优先级的中断同时产生，那么高优先级的中断将被首先处理。

6.2.3 查询顺序

如果两个相同优先级的中断同时出现，将由固定的查询顺序来决定首先响应哪

一个中断。该查询顺序是：外部中断0、定时器0、外部中断1、定时器1、串行端口、定时器2。

图6-2描述了5个中断源的独立和全局启用机制、查询顺序及优先级设置。所有中断由特殊功能寄存器的相应标志位置位来激活。当然，如果某个中断被禁用，则该中断不再发生，但是软件仍然可以测试相应的中断标志位。前两章就使用了定时器和串行端口的中断标志，但其实并未用到中断。

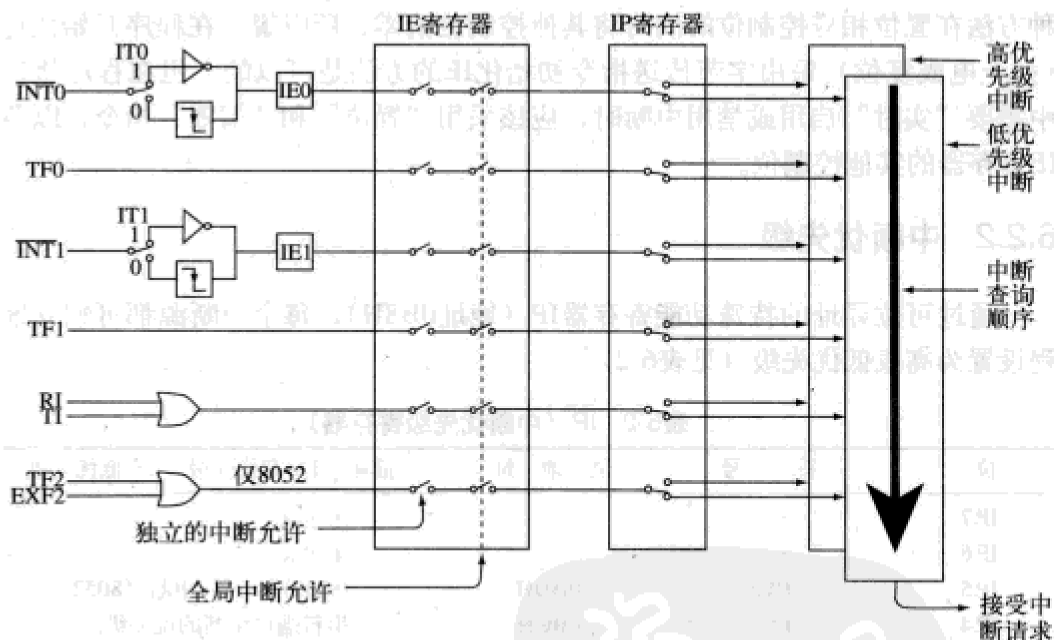


图6-2 8051中断结构

串行端口中断可由接受中断RI或者发送中断TI触发。同样地，定时器2中断可由定时器溢出标志位TF2或外部输入标志EXF2触发。表6-3列出了所有能够触发中断的标志位。

134～135

表6-3 中断标志位

中 断	标 志	所在的特殊功能寄存器及位地址
外部0	IE0	TCON.1
外部1	IE1	TCON.3
定时器1	TF1	TCON.7
定时器0	TF0	TCON.5
串行端口	TI	SCON.1
串行端口	RI	SCON.0
定时器2	TF2	T2CON.7 (8052)
定时器2	EXF2	T2CON.6 (8052)

6.3 中断处理

当某中断产生而且被CPU响应，主程序被中断，接下来将执行如下操作：

- 当前正被执行的指令全部执行完毕；
- PC值被压入栈；
- 现场保护；
- 阻止同级别其他中断；
- 将中断向量地址装载到程序计数器PC；
- 执行相应的中断服务程序。

中断服务程序ISR完成和该中断相应的一些操作。ISR以RETI（中段返回）指令结束，将PC值从栈取回，并恢复原来的中断设置，之后从主程序的断点处继续执行。

中断向量

当某中断被响应时，被装载到程序计数器PC中的数值称为**中断向量**，是同该中断源相对应的中断服务程序的起始地址。表6-4给出了所有中断源的中断向量。

系统复位向量（RST位于地址0000H）也被列在表中，因为它可以打断主程序并将程序计数器赋予新值，在此意义上，RST也类似1个中断。

当“转去执行中断”时，引起中断的标志位将被硬件自动清零，但串行端口中断的RI和TI以及定时器2的TF2和EXF2除外。由于它们每个中断都有2个可能的中断源，所以靠CPU来清零中断标志是不可行的。这些标志位需要通过ISR来测试究竟是哪一个中断源引发的中断，然后再通过软件将其清零。一般情况下程序会根据中断源的不同转到不同的分支去执行相应的操作。

由于中断向量位于程序存储器的底部，所以主程序的第1条指令通常为跳转指令，越过中断向量区，例如LJMP 0030H。

表6-4 中断向量

中 断	标 志	向量地址
系统复位	RST	0000H
外部0	IE0	0003H
定时器0	TF0	000BH
外部1	IE1	0013H
定时器1	TF1	001BH
串行端口	RI或TI	0023H
定时器2	TF2或EXF2	002BH

6.4　中断程序设计

在第3章和第4章中的一些例子中，虽然没有用到中断，但是大量使用了“等待循环”来测试定时器溢出标志（TF0、TF1或TF2）或串行发送和接收标志（TI和RI）。在这种方式中存在一个问题，就是CPU的宝贵执行时间都消耗在了等待该标志被置位的过程中，而不能执行其他任务。这对于一个面向控制的微控制器系统来说显然是不合适的，因为需要系统同时和很多输入和输出设备打交道。

在本节中，将通过一些面向控制应用的例子来研究实际过程中软件的实现方法。其中的关键因素就是中断。虽然这些程序不一定很大，但较复杂，因此为了理解这些程序，我们需要一步接一步地仔细分析。建议读者耐心仔细地琢磨这些例子，中断是系统设计中最难解决的问题之一，这些程序的一些细节必须彻底弄明白。

由于用到了中断，所以例程都是完整的。每个程序都开始于0000H，也就是假设其在系统复位后被开始执行。意思就是说，这些程序都已开发调试完成并已经下载到ROM或EPROM中，可以马上正常运行。

应用中断的完整程序的框架推荐如下：

```
        ORG   0000H      ;RESET ENTRY POINT
        LJMP  MAIN
        .                ;ISR ENTRY POINTS
        .
        .
        .
        ORG 0030H        ;MAIN PROGRAM ENTRY POINT
MAIN:   .                ;MAIN PROGRAM BEGINS
        .
        .
```

第1条指令跳转到地址0030H，刚好跳过了中断向量占用的存储空间（见表6-4），如图6-3所示，主程序从地址0030H开始。



图6-3　在使用中断情况下的程序存储器组织

6.4.1 小型中断服务例程

如表6-4所示，中断服务例程必须开始于程序存储器的底部。虽然在两个相邻的中断向量之间只有8B的存储空间，但通常可以满足中断服务程序中的指令对存储空间的需要。

如果只有一个定时器0中断源被使用，那么可使用下面的程序架构：

137

```
ORG         0000H          ;RESET
LJMP        MAIN
ORG         000BH          ;Timer 0 ENTRY POINT
T0ISR:      .              ;Timer 0 ISR BEGINS
            .
            RETI           ;RETURN TO MAIN PROGRAM
MAIN:       .              ;MAIN PROGRAM
            .
            .
```

如果使用多个中断，必须确保正确设置每个中断的入口地址（见表6-4），并使ISR不要超过8B，否则将覆盖下一个中断程序的内容。由于上面的例子中只使用了1个中断，所以主程序可以紧随RETI指令之后开始。

6.4.2 大型中断服务例程

如果中断服务例程的长度超过8B，那么必须被放置到程序存储器的其他地址（跳过中断向量区）或者侵占下一个中断的空间。一般情况下，对较大ISR都采取放置到其他存储区域的方案。考虑只使用定时器0中断的情形，可采用如下的程序结构：

```
            ORG     000H     ;RESET ENTRY POINT
            LJMP    MAIN
            ORG     000BH    ;Timer 0 ENTRY POINT
            LJMP    T0ISR
            ORG     0030H    ;ABOVE INTERRUPT VECTORS
MAIN:       .
            .
            .
T0ISR:      .
            .                ;Timer 0 ISR
            .
            RETI             ;RETURN TO MAIN PROGRAM
```

138

为了简单起见，只让程序在起始处做一件事情。主程序首先对定时器、串行端口和中断寄存器做相应的初始化，之后就不再做什么了，因为需要执行的任务都放在了中断服务例程里。初始化指令之后，主程序包含如下指令：

```
HERE:     SJMP HERE
```

当中断发生时，主程序被暂时打断，转去执行中断服务程序。位于ISR结尾处的RETI指令将控制权交还给主程序，主程序仍然回到原地踏步等待状态。这并非牵强的想象，事实上，在许多面向控制的应用系统中，许多任务就是由中断例程完成的。

6.5 定时器中断

当定时器寄存器THx/TLx发生溢出、溢出标志TFx被置位时，定时器中断发生。8051转去处理该中断，此时定时器的溢出标志TFx被硬件清除。因此，在中断被启用的情况下，TFx将被硬件自动清除，而在第4章中，因为没有用到中断，所以需要应用软件指令来清除溢出标志。

例6-1 应用定时器中断产生方波

编写程序，利用定时器0中断在P1.0产生10kHz的方波。

答案：

```
0000            5           ORG   0        ;reset entry point
0000 020030     6           LJMP  MAIN     ;jump above interrupt vectors
000B            7           ORG   000BH    ;Timer 0 interrupt vector
000B B290       8   T0ISR:  CPL   P1.0     ;toggle port bit
000D 32         9           RETI
0030           10           ORG   0030H    ;Main program entry point
0030 758902    11   MAIN:   MOV   TMOD,#02H ;timer 0, mode 2
0033 758CCE    12           MOV   TH0,#-50  ;50 us delay
0036 D28C      13           SETB  TR0       ;start timer
0038 75A882    14           MOV   IE,#82H   ;enable timer 0 interrupt
003B 80FE      15           SJMP  $         ;do nothing
```

讨论： 当定时器中断启用后，定时器寄存器THx/TLx发生溢出将使标志位TFx置位，从而引发中断。这个例子在第4章中出现过，区别是那时没有使用中断，所以除了用到了中断以外，程序的其余部分是相同的。

上面的程序是一个完整的程序，其可以烧录到EPROM芯片中，然后再安装到8051单板机上执行。复位之后程序计数器PC的值被更新为0000H，执行第1条指令（LJMP MAIN），越过定时器中断服务程序跳转到程序存储器的0030H地址。接下来的3条指令（11行~13行），将定时器0初始化为8位自动重装载模式，每隔50μs溢出1次。第14行的MOV指令启用定时器0溢出中断，所以定时器每溢出1次就产生1次中断。当然，第1次溢出要发生在50μs之后，所以，在主程序的最后安排1条什么也不做的循环等待指令。每50μs中断1次，主程序被打断，转去执行定时器0中断的服务程序。ISR对相应的端口位取反（第8行）之后即返回到主程序（第9行）继续进行原地循环，等待50μs之后下一次中断发生。

注意，定时器0的溢出标志TF0无需软件清除。当相应的中断被启用后，TF0在CPU转去处理中断时自动由硬件清除。

由于答案是完整程序，所以我们必须知道在此过程中栈是如何操作的。中断服务例程的返回地址是SJMP指令位于程序存储器中的地址。8051将转去执行ISR之前运行指令的地址压入栈，当执行RETI指令时（第9行），该地址再从栈中弹出。由于在程序中没有初始化栈指针SP，默认值是07H。上述被压入栈中的返回地址保存在内部RAM中的08H（PC_H）和09H（PC_L）。

139

例6-2 利用中断产生2个方波

编写程序，利用中断分别在P1.7和P1.6同时产生频率为7kHz和500Hz的方波。

答案：图6-4是实现双方波同时输出的硬件配置图。

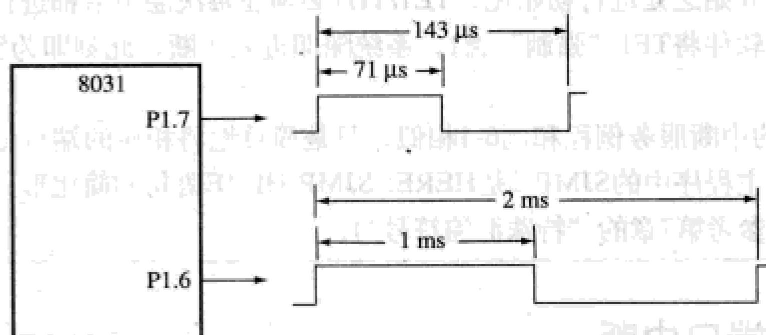


图6-4 波形实例

题目要求的任务如果在非中断系统中实现是非常困难的。像例6-1一样，由定时器0工作在模式2来产生7kHz的方波，由定时器1工作在模式1（16位定时器模式）来产生500Hz的方波，高电平和低电平的持续时间均为1ms，由于定时间隔较长，所以最好不要采用工作模式2（8051工作在12MHz时钟频率时最大定时间隔为256ms）。下面是程序：

140

```
0000            5           ORG   0
0000   020030   6           LJMP  MAIN
000B            7           ORG   000BH                ;Timer 0 vector address
000B   02003F   8           LJMP  T0ISR
001B            9           ORG   001BH                ;Timer 1 vector address
001B   020042   10          LJMP  T1ISR
0030            11          ORG   0030H
0030   758912   12  MAIN:   MOV   TMOD,#12H            ;Timer 1 = mode 1
                13                                     ;Timer 0 = mode 2
0033   758CB9   14          MOV   TH0,#-71             ;7 kHz using timer 0
0036   D28C     15          SETB  TR0
0038   D28F     16          SETB  TF1                  ;force timer 1 interrupt
003A   75A88A   17          MOV   IE,#8AH              ;enable both timer interrupts
003D   80FE     18          SJMP  $
                19  ;
003F   B297     20  T0ISR:  CPL   P1.7
0041   32       21          RETI
0042   C28E     22  T1ISR:  CLR   TR1
0044   758DFC   23          MOV   TH1,#HIGH(-1000);1 ms high time &
0047   758B18   24          MOV   TL1,#LOW(-1000) ; low time
004A   D28E     25          SETB  TR1
004C   B296     26          CPL   P1.6
004E   32       27          RETI
                28          END
```

讨论：该程序是完整的，编译后可以直接下载到基于8051产品的EPROM或ROM中运行。其中的主程序和2个ISR是跳过了复位和中断向量空间存放的，两种方波都由“CPL bit”指令生成，但二者的周期有所不同。

由于TL1/TH1寄存器在每次溢出后都需要重新装载计数初值（也就是在每次中断之后），在定时器1的中断服务例程中，(a) 停止定时器，(b) 重载TL1/TH1，(c) 启动定时器，(d) 翻转相应的端口位。注意，此处的TL1/TH1不像定时器0那样在主程序的开始之处进行初始化，TL1/TH1必须在每次溢出后都进行重载。在主程序中，通过软件将TF1"强制"置1，系统随即进入中断，此刻即为500Hz方波的起始时刻。

定时器0的中断服务例程和例6-1相似，只是简单地将相应的端口位取反而后即返回主程序。主程序中的SJMP $是HERE: SJMP HERE语句的简化形式，二者的功能是等效的（参考第7章的"特殊汇编符号"）。

141

6.6　串行端口中断

当发送中断标志TI或接受中断标志RI被置位时，将引发串行端口中断。其中发送中断出现在SBUF寄存器中的字符被发送完毕的时刻；接收中断发生在1个字符被完全接收存放到SBUF中等待软件读取的时候。

串行端口中断和前面讨论的定时器中断有所不同，当CPU转而去处理中断事件时，中断标志位不能由硬件清除，其原因是串行端口中断有2个中断源：TI和RI。究竟是哪个中断源，需要在ISR中通过软件判定，并且要用软件方式清零相应的中断标志位。定时器溢出中断的标志位是在CPU转向ISR时由硬件清除的。

例6-3　利用中断实现字符输出

编写程序，采用中断方式通过8051的串行口连续向外部终端发送ASCII码表（控制码除外）。

答案：ASCII码表共有128个7位的ASCII码（参考附录F）。包括95个图形码（20H到7EH）和33个控制码（00H到1FH，以及7FH）。下面的程序也是完整的汇编程序，可以编译下载到系统的EPROM或ROM中，复位后即可正常运行。

```
0000              5              ORG     0
0000    020030    6              LJMP    MAIN
0023              7              ORG     0023H        ;serial port interrupt entry
0023    020042    8              LJMP    SPISR
0030              9              ORG     0030H
0030    758920    10    MAIN:    MOV     TMOD,#20H    ;Timer 1, mode 2
0033    758DE6    11             MOV     TH1,#-26     ;12000 baud reload value
0036    D28E      12             SETB    TR1          ;start timer
0038    759842    13             MOV     SCON,#42H    ;mode 1, set TI to force 1st
                  14                                  ; interrupt; send 1st char.
003B    7420      15             MOV     A,#20H       ;send ASCII space first
003D    75A890    16             MOV     IE,90H       ;enable serial port interrupt
0040    80FE      17             SJMP    $            ;do nothing
                  18    ;
0042    B47F02    19    SPISR:   CJNE    A,#7FH,SKIP  ;if finished ASCII set,
```

```
0045   7420   20          MOV    A,#20H    ; reset to SPACE
0047   F599   21  SKIP:   MOV    SBUF,A    ;send char. to serial port
0049   04     22          INC    A         ;increment ASCII code
004A   C299   23          CLR    TI        ;clear interrupt flag
004C   32     24          RETI
              25          END
```

讨论：跳转到MAIN所在的地址0030H之后，安排了3条指令来完成定时器1的初始化工作，为串行端口提供1200的波特率时钟（第10~12行）。MOV SCON，#42H语句选择串行端口工作模式1（8位的UART），同时将TI置位，强制产生1次串行中断。然后，将第1个ASCII码（20H）传送到累加器A，并且打开串行端口中断。最后，主程序进入原地踏步的等待循环（SJMP $）。

142

一旦主程序完成上述一系列初始化设置之后，所有的任务都由中断服务例程完成。前2条指令检测累加器A的内容，如果其中的ASCII码为7FH（就是说发送的最后1个ASCII码为7EH），那么将累加器的内容重新设置为20H（第19~20行）。然后，将存储在累加器中的ASCII码发送到串行端口缓冲寄存器（MOV SBUF，A），之后，累加器的内容加1（INC A），清零发送中断标志（CLR TI），最后中断服务例程结束（RETI）。控制权交还给主程序，继续执行SJMP $，直到字符发送完毕时TI被置1再次触发中断。

如果比较一下CPU的运行速度和串行端口发送字符的速率，可知SJMP $指令的执行时间在整个程序执行时间中占据了很大的份额。那么，到底占多少百分比呢？对于1200波特率，每1位的发送时间为1/1200＝0.833ms，8位数据加上起始位和结束位共需要8.33ms，即8333μs。考虑最坏的情况，SPJSR的执行时间为所有指令所占总机器周期数乘以1μs（假设使用12MHz晶振），结果是8μs。所以，在每个字节的8333μs的发送时间中ISR的运行只需8μs，而SJMP $语句占用的时间百分比为8325/8333×100%＝99.9%。因此，在使用中断的情况下，可以在具体应用中考虑用别的指令代替SJMP $来同时完成其他任务。如同本例一样，中断的情况不变，还是8.33ms产生1次，仍然以1200波特率通过串行端口连续发送字符数据。

6.7 外部中断

当CPU在8051芯片的 $\overline{INT0}$ 和 $\overline{INT1}$ 引脚探测到低电平或下降沿时，产生外部中断。$\overline{INT0}$ 和 $\overline{INT1}$ 分别是端口3的位P3.2（12脚）和P3.3（13脚）。

触发2个外部中断的标志位是位于寄存器TCON中的IE0和IE1。当外部中断被触发时，中断标志位仅在边沿触发情况下由硬件清除，如果是低电平触发，那么中断标志位由外部中断源控制而非内部硬件。

通过编程设置寄存器TCON的IT0和IT1位来选择低电平触发和下降沿触发。例

如，如果IT1＝0，那么系统在 $\overline{\text{INT1}}$ 引脚探测到低电平信号后产生外部中断1。如果IT1＝1，那么外部中断1为边沿触发方式，如果在 $\overline{\text{INT1}}$ 的前后2次采样，其中前1个周期为高电平，而后1个周期为低电平，那么置位中断请求标志位IE1（位于TCON），标志位IE1向CPU提出中断请求。

由于系统每个机器周期对外部中断引脚采样1次，所以为了确保被检测到，输入信号应该至少维持12个振荡周期。如果外部中断为下降沿触发，要求必须在相应的引脚维持高电平至少1个机器周期，而且低电平也要持续大于1个机器周期，才能确保该下降沿被CPU检测到。在CPU转去执行ISR之后，IE0和IE1由硬件自动清除。

143

如果外部中断为电平触发，外部中断源应保持低电平直至CPU接受中断请求。在中断服务例程执行完毕之前，该低电平必须撤销，否则将再次引发中断。通常，在ISR中设置一些操作使得外部中断源恢复为高电平。

例6-4　锅炉控制

利用中断方式，设计1个锅炉控温系统，要求温度保持在（20±1）°C。

答案：假设本例的硬件接续如下，锅炉的开关线圈和P1.7相连，即

P1.7=1 对应线圈接通（锅炉打开）；

P1.7=0 对应线圈断开（锅炉关闭）。

温度传感器连接在 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$，分别提供 $\overline{\text{HOT}}$（加热）和 $\overline{\text{COLD}}$（制冷）信号，即

若 T >21°C，则 $\overline{\text{HOT}}$=0，

若 T <19°C，则 $\overline{\text{COLD}}$=0。

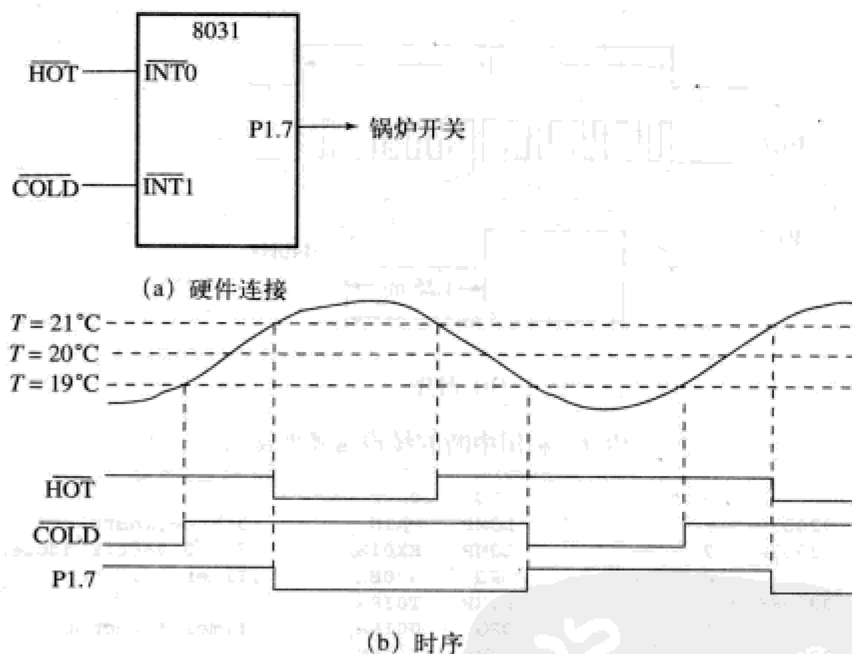
程序应该在 T<19°C时启动锅炉加热装置，在 T>21°C时停止锅炉加热装置。该系统硬件配置和时序图如图6-5所示。

```
0000              5             ORG    0
0000    020030    6             LJMP   MAIN
                  7                           ;EXT 0 vector at 0003H
0003    C297      8   EX0ISR:   CLR    P1.7   ;turn furnace off
0005    32        9             RETI
0013              10            ORG    0013H
0013    D297      11  EX1ISR:   SETB   P1.7   ;turn furnace on
0015    32        12            RETI
0030              13            ORG    30H
0030    75A885    14  MAIN:     MOV    IE,#85H       ;enable external interrupts
0033    D288      15            SETB   IT0           ;negative edge triggered
0035    D28A      16            SETB   IT1
0037    D297      17            SETB   P1.7          ;turn furnace on
0039    20B202    18            JB     P3.2,SKIP     ;if T > 21 degrees,
003C    C297      19            CLR    P1.7          ; turn furnace off
003E    80FE      20  SKIP:     SJMP   $             ;do nothing
                  21            END
```

讨论：主函数的前3条指令（第14~16行）打开外部中断，并将 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ 都设为下降沿触发方式。由于当前的 $\overline{\text{HOT}}$（P3.2）和 $\overline{\text{COLD}}$（P3.3）的输入状态未知，所

以接下来的3条指令（第17~19行）需要合理地确定是应该打开还是关闭锅炉。首先打开锅炉（SETB P1.7），然后采样 $\overline{\text{HOT}}$ 信号（JB P3.2，SKIP），如果 $\overline{\text{HOT}}$ 为高，表示T<21°C，所以下一条指令被跳过继续保持加热状态。如果，$\overline{\text{HOT}}$ 为低，表示T>21°C，不再跳过而是执行下一条指令，关闭锅炉加热装置（CLR P1.7），进入原地循环状态，等待中断发生。

144



（a）硬件连接

（b）时序

图6-5 锅炉控制

一旦主程序完成了合理的设置，之后就无需再做什么了。每次当温度超过21°C或低于19°C时，就会产生相应的中断，中断服务程序会合理地打开（SETB P1.7）或关闭锅炉（CLR P1.7），然后返回主程序。

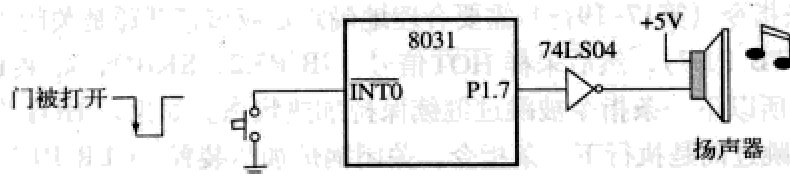
注意，本例中，在标号EX0ISR之前无需再添加ORG 0003H指令，因为LJMP MAIN指令的长度为3字节，所以EX0ISR标号的地址为0003H，恰好是外部中断0的入口地址。

例6-5 入侵报警系统

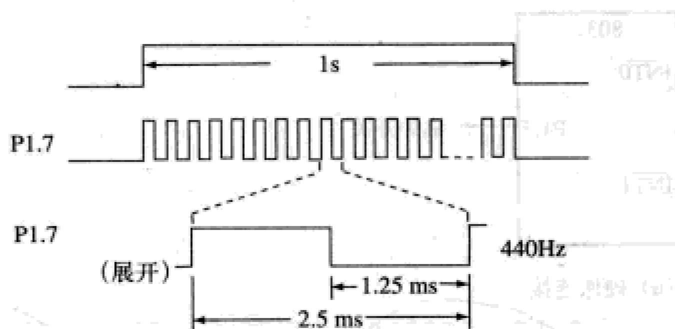
设计一个入侵报警系统，要求：当连接在 $\overline{\text{INT0}}$ 引脚的门禁传感器产生从高到低的电平跳变时，通过扬声器发出400Hz持续1s的报警声（扬声器接在P1.7）。

145

答案：本解决方案用到了3个中断：外部中断0（门禁传感器）、定时器0中断（1s延时）和定时器1中断（400Hz报警声）。系统的硬件配置和运行时序如图6-6所示。



(a) 硬件连接



(b) 时序

图6-6　利用中断的扬声器硬件接口

```
0000              5                 ORG      0
0000   020030     6                 LJMP     MAIN          ;3-byte instruction
0003   02003A     7                 LJMP     EX0ISR        ;EXT 0 vector address
000B              8                 ORG      000BH         ;Timer 0 vector
000B   020045     9                 LJMP     T0ISR
001B              10                ORG      001BH         ;Timer 1 vector
001B   020059     11                LJMP     T1ISR
0030              12                ORG      0030H
0030   D288       13     MAIN:      SETB     IT0           ;negative edge activated
0032   758911     14                MOV      TMOD,#11H     ;16-bit timer mode
0035   75A881     15                MOV      IE,#81H       ;enable EXT 0 only
0038   80FE       16                SJMP     $             ;now relax
                  17     ;
003A   7F14       18     EX0ISR:    MOV      R7,#20        ;20 ' 5000 us = 1 second
003C   D28D       19                SETB     TF0           ;force timer 0 interrupt
003E   D28F       20                SETB     TF1           ;force timer 1 interrupt
0040   D2A9       21                SETB     ET0           ;begin tone for 1 second
0042   D2AB       22                SETB     ET1           ;enable timer interrupts
0044   32         23                RETI                   ;timer ints will do the work
                  24     ;
0045   C28C       25     T0ISR:     CLR      TR0           ;stop timer
0047   DF07       26                DJNZ     R7,SKIP       ;if not 20th time, exit
0049   C2A9       27                CLR      ET0           ;if 20th, disable tone
004B   C2AB       28                CLR      ET1           ;disable itself
004D   020058     29                LJMP     EXIT
0050   758C3C     30     SKIP:      MOV      TH0,#HIGH(-50000)    ;0.05 sec. delay
0053   758AB0     31                MOV      TL0,#LOW(-5000)
0056   D28C       32                SETB     TR0
0058   32         33     EXIT:      RETI
                  34     ;
0059   C28E       35     T1ISR:     CLR      TR1
005B   758DFB     36                MOV      TH1,#HIGH(-1250)     ;count for 400 Hz
005E   758B1E     37                MOV      TL1,#LOW(-1250)
```

```
0061    B297    38          CPL    P1.7        ;music maestro!
0063    D28E    39          SETB   TR1
0065    32      40          RETI
                41          END
```

讨论：这是迄今为止本书中最长的程序。程序分为5个部分：中断向量声明，主程序，3个中断服务例程。所有向量空间都包含1条LJMP指令跳转到各自对应的ISR。主程序开始于程序存储器的地址0030H处，只有4条指令。SETB IT0将门禁传感器的中断输入方式设置为下降沿触发。MOV TMOD，#11H将2个定时器设置为模式1，即16位定时器模式。在主程序中仅打开了外部中断0（MOV IE，#81H），所以“门打开”是CPU可以接受所有中断的前提条件。最后，SJMP $使主程序进入原地循环状态。

当1个门打开信号被探测到时（$\overline{INT0}$引脚出现高到低的电平跳变），触发外部中断0，在EX0ISR的开始首先在R7中放置常数20（见下文）。然后将2个定时器的溢出标志位置1强制产生定时器中断。

不过想要产生定时器中断，还需要在IE寄存器中将相应的中断允许位置1才行，接下来的2条指令（SETB ET0和SETB ET1）即能实现此功能。最后，EX0ISR以RETI（中断返回指令）结束。

定时器0产生1s的定时间隔，定时器1产生400Hz的报警声。当从EX0ISR返回至主程序之后立即产生定时器中断（在CPU执行一次SJMP $ 指令之后）。由于程序中没有进行中断优先级的设置，所以系统按默认的查询顺序来处理发生的中断，也就是说首先接受定时期0中断。通过将50000μs的延时重复20次的方法得到1s的延时，R7作该计数器。其中的第19次重复的时候，T0ISR的执行过程如下。首先，定时器0计数停止，R7内容减1。然后，TH0/TL0重新载入计数初值−50 000，启动定时器0，中段结束。接着是第20次定时器0中断，R7的内容减到了0（1s的延时已经完成），2个定时器中断都被禁用（CLR ET0和CLR ET1），中断结束。此时，定时器不再产生中断请求，直至再次检测到“门打开”信号。

147

400Hz的报警声被编程为由定时器1中断来实现。400Hz方波的周期为1/400＝2500μs，其中高电平时间为1250μs，低电平时间也为1250μs。每次执行定时器1的ISR所进行的操作比较简单，首先把−1250重载到TH1/TL1，然后将驱动扬声器的端口位取反，中断结束。

6.8 中断时序

系统在每个机器周期的S5P2对中断进行采样和锁存（如图6-7所示），在下一个机器周期查询所有中断源的状态，如果中断4条件存在，而且满足以下3个条件，该中断才能被系统接受：（a）系统目前没有处理其他同优先级或高优先级的中断；（b）查询周期是当前执行指令的最后1个周期；（c）当前指令不是RETI或任何访问

IE和IP的指令。在随后的一个机器周期中，CPU将当前PC的内容压入栈，然后把中断向量地址载入到PC中，ISR开始执行。

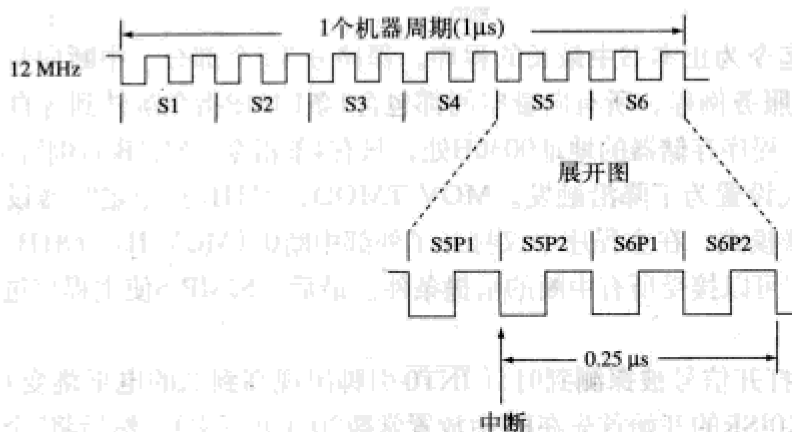
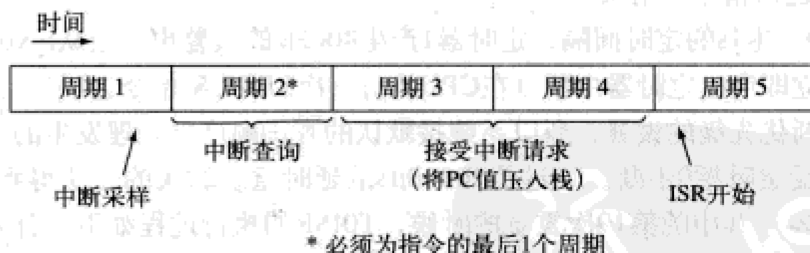


图6-7　在S5P2期间的中断采样

上面的约束条件（c）是为了确保在每个中断服务例程之后至少再执行1条指令，中断查询的时序如图6-8所示。



148

图6-8　中断的查询

从发出中断请求到相应的ISR开始执行之间的这段时间称为**中断等待时间**。在某些面向控制应用的场合，中断等待时间是影响系统响应速度的。对于12MHz的晶振，8051的中断等待时间最短为3.25μs。在最坏情况下，8051系统的1级（高优先级）中断的中断等待时间可达9.25μs（假设高优先级中断是允许的）。如图6-9所示，这种情况是：中断条件产生于0级（低优先级）中断服务程序的RETI指令之前，而且返回主程序后执行的第1条指令是乘法指令。

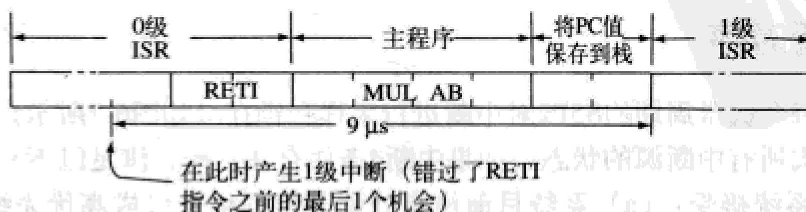


图6-9　中断等待时间

小结

本章主要介绍了基于8051微控制器中断系统的应用设计过程中的一些细节问题。建议读者逐渐熟悉中断编程，本章的例程都是初学者理解8051中断编程的好例子。

8051单板机通常在EPROM的底部驻留1段监视程序。如果在监视程序中没有用到中断，那么在中断向量空间可能安排有1条LJMP指令，指向代码RAM的某个区域，在这个区域存放着可以进行执行或调试的用户应用程序。制造商通常向用户提供中断向量的入口地址，以便程序员用作中断服务程序的入口。用户也可以用监视程序提供的命令直接查找中断向量空间的内容。例如，程序存储器地址0003H单元的内容是系统接受外部中断0的中断请求后执行的第1条指令的操作码，如果是LJMP操作码（22H，参考附录B），则随后的两个地址（0004H和0005H）为该中断服务程序的地址，以此类推。

另一个方案是用户开发完整的中断应用程序，正如本章中的一些例子所示。例程的目标代码可以烧录到EPROM然后装载到目标系统上，代码地址为0000H。当系统上电或者复位时，应用程序即可自动执行，无需监视程序。

习题

6.1　修改例6-1，使得如果按下终端的任意键即可禁止中断并结束程序。

6.2　利用中断在P1.7产生1kHz的方波。

6.3　采用中断在P1.6产生7kHz、占空比30%的脉冲信号。

6.4　将例6-1和例6-2的程序合成为1个程序。

6.5　修改例6-3中的程序，实现每秒发送1个字符。（提示：利用定时器，在其中断服务例程中发送字符。）

6.6　修改例6-5中的程序，增加1个“再启动”模式，当扬声器正在鸣响时，如果在 $\overline{\text{INT0}}$ 出现由高到低的电平跳变，那么重新启动延时循环产生新1s的鸣叫。请参考图6-10。

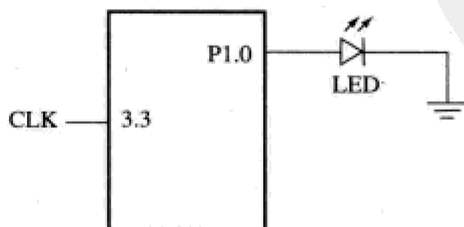


图6-10　使用中断的LED接口

6.7　假设外部中断0（连接到INT0）和定时器0溢出中断同时发生，请问哪个中断源将

被系统首先响应？为什么？

6.8　默认情况下，当串行端口中断和外部中断1同时出现时，系统将首先响应外部中断1，然后才响应串行端口中断，如何设置8051才能将二者的先后顺序颠倒过来？

6.9　寄存器IE和IP被初始化为：IE = 10001111，IP = 00001110，假设定时器0溢出中断、串行端口中断和外部中断1同时产生，请问哪个中断先被响应？为什么？

6.10　小型中断服务例程和大型中断服务例程的区别是什么？

6.11　参考图6-10，假设P3.3接有1kHz的时钟信号，编写一段汇编程序，启用外部中断1，当系统在P3.3探测到时钟信号的由高到低的电平跳变时，置位P1.0（点亮LED），并持续大约250μs。

150



第7章 汇编语言编程

7.1 引言

本章介绍8051微控制器的汇编语言编程。汇编语言是介于机器语言和高级语言这两种截然不同的语言之间的计算机语言。典型的高级语言（如Pascal、C语言等）使用易于被人理解的关键字和语句来编写，当然它同自然语言还是存在很大差别的。机器语言是计算机的二进制语言，机器语言程序是一系列计算机可执行的代表相应指令的二进制字节。

汇编语言以便于记忆的“助记符”替代了机器语言的二进制代码，从而简化了程序设计。例如，加法指令在机器语言里可以表示为二进制代码“10110011”；在汇编语言里可以表示为助记符“ADD”。使用助记符编程显然比使用二进制码更可取。

当然，这不是编程工作的全部。指令要操作数据，数据的存储位置由嵌在指令机器语言二进制码里的各种“寻址模式”来规定。所以，根据被加数的形式，存在多种加法指令。这些关于变量的规则和约定是本章的学习重点。

计算机不能执行汇编语言程序，必须翻译成机器语言才能在计算机上执行。仍然看上文的例子，助记符“ADD”必须翻译成二进制码“10110011”才能被计算机识别。随编程环境的复杂程度不同，将汇编程序翻译成可执行的机器语言程序可能需要一个或多个步骤。在将指令助记符翻译成机器语言二进制代码的过程中，“汇编器”程序是必不可少的。更进一步可能还需要一个“链接器”，用于将离散的文件作为程序的一个部分集成在一起，并且设定程序执行时在内存中的地址。下面给出几个基本定义。

汇编语言程序是用标号、助记符等编写而成的程序，每条语句对应一个机器指令。汇编语言通常被称为源代码或符号代码，不能在计算机上执行。

机器语言程序由代表指令的二进制代码组成，机器语言程序通常被称为目标代码，可以在计算机上执行。

汇编器是一个能够将汇编语言程序翻译成机器语言程序的程序。机器语言程序（目标代码）既可以采用绝对地址，也可以是能重新定位的。如果是后者，那么需用“链接”操作来确定程序执行时的绝对地址。

链接器是一个程序，用于将可重新定位目标程序（模块）链接起来并且产生具

有绝对地址的、可以在计算机上执行的目标程序。链接器有时被称为“链接/定位器”，意思是先将可重新定位的模块组合（链接）起来，然后再设置其可执行地址（定位）。

段是数据或程序存储器的一个单元。段可以是可重新定位的，也可以是有绝对地址的。可重新定位的段具有名称、类型和其他属性，允许链接器将其整合到其他段之中，如果需要还可以将该段重新定位。与此相反，绝对地址段没有名称，而且不能被整合到其他段。

模块由一个或多个段组成，模块有自己的名称，可由用户定义。定义1个模块也就决定了本地符号的有效范围。一个目标文件包含一个或多个模块，在一些场合下可以将模块视为“文件”来理解。

程序包含一个具有绝对地址的模块，该模块是由所有输入模块的绝对地址段和可重定位段整合而成的。程序仅包含能被计算机识别的、和指令（带有地址和数据常数）对应的二进制代码。

7.2 汇编器操作

多种汇编程序及其支持程序的存在客观上促进了8051微控制器应用技术的发展。虽然Intel公司最早的MCS-51™系列汇编器——ASM51™已不再商用，但它建立了供其他公司遵守的行业标准。本章着重讨论具有ASM51共性特征的汇编语言程序，尽管许多特性已经标准化，但有可能在某些其他汇编器中不能执行。

ASM51是配置齐备、功能强大的汇编器，可以应用于Intel的开发系统及IBM系列个人微机中。由于包含CPU的主机不同于8051微控制器，所以ASM51被称为**交叉汇编器**。8051源程序可在主机上编写（使用文本编辑器），然后汇编成目标文件和列表文件（使用ASM51），但程序还不能在主机上直接执行。因为主机系统的CPU不同于8051，它不能识别目标文件中的二进制指令。在主机上执行目标文件需要借助于目标CPU的硬件仿真或软件模拟才能实现。第3种可能是将目标文件下载到基于8051的目标系统之中执行。硬件仿真、软件模拟、下载到目标系统及其他开发技术将在第10章讨论。

152

在系统提示符下输入如下命令，即可调用ASM51：

```
ASM51 source_file  [assembler_controls]
```

该命令完成源文件的汇编而且性能受汇编器控制选项的影响（汇编器控制选项是可选的，将在本章的后续部分讨论）。汇编器受理输入的1个源文件（PROGRAM.SRC），输出1个目标文件（PROGRAM.OBJ）和1个列表文件（PROGRAM.LST）。该汇编过程如图7-1所示。

由于大多数汇编器在将汇编语言翻译成机器语言的过程中需要两次扫描源文件，所以被称为**两次扫描汇编器**。汇编器用**定位计数器**作为指令的地址和标号地址

的数值。下面分别介绍每次扫描的过程。

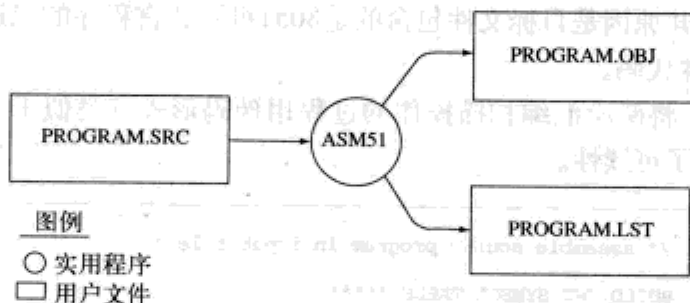


图7-1 源文件的汇编过程

7.2.1 第1次扫描

在第1次扫描过程中，汇编器逐行扫描源文件并建立符号表。定位计数器的默认值为0或者由ORG伪指令设定。当扫描源文件时，定位计数器的数值按指令的长度递增。需要特别指出的是，当遇到定义数据的伪指令（DB和DW）时，定位计数器的增量等于定义指令所定义的数据的字节数；当遇到预留存储器伪指令（DS）时，定位计数器的增量等于DS指令所预留的字节数。

每次在1行指令的开头发现标号时，汇编器会将该标号和定位计数器数的当前值一同添加到符号表中。采用EQU伪指令定义的符号，连同求出的数值一并添加到符号表里。第1次扫描结束时，保存符号表以供第2次扫描使用。

7.2.2 第2次扫描

在第2次扫描过程中，汇编器建立目标文件和列表文件。助记符被转换为操作码之后添加到输出文件中。汇编器对操作数表达式求值，把结果放置在指令操作码后面。如果符号出现在操作数字段里，那么需要从第1次扫描所建立的符号表中找到其所对应的数值，并用来计算该指令的正确的数据或地址。

153

由于要执行两次扫描，源程序中可能会使用“前向引用”，也就是说，1个符号在定义之前就被使用了。例如，前向跳转就是这种情况。

如果目标文件是有绝对地址的，那么仅包括机器语言程序的二进制字节（00H~FFH）。如果目标文件是可重新定位的，那么除此之外还包括符号表和链接定位所需要的相关信息。源程序和机器语言程序中十六进制字节的列表文件都由ASCII码表示（20H~7EH）。

如果在主机上分别打开目标文件和列表文件（例如在MS-DOS系统中利用TYPE命令），可以很清楚地看到二者的区别。列表文件可以完好地显示，在程序语句（来自于源程序）之后，输出的每1行包括地址、操作码，或许还有数据。列表文件

可以完好显示是由于其只包含ASCII文本代码。当显示目标文件时会出现问题，会输出一些乱码，其原因是目标文件包含的是8051机器语言程序的二进制代码，而非ASCII格式的文本代码。

在图7-2中，将两次汇编扫描操作的过程用伪码形式（类似于Pascal或C语言）表述出来，增加了可读性。

```
ASM(input_file)  /* assemble source program in input_file */
BEGIN
  /***** PASS 1: BUILD THE SYMBOL TABLE *****/
  lc = 0;      /* lc = location counter */
  mnemonic = null;
  open_input_file;
  WHILE (mnemonic != end) DO BEGIN
    get_line();
    scan_line();  /* get label/symbol and mnemonic */
    IF (label) THEN
      enter_in_symbol_table(label, lc);
    CASE mnemonic OF BEGIN
      null, comment, end: ; /* do nothing */
      ORG: lc = operand_value;
      EQU: enter_in_symbol_table(symbol, operand_value);
      DB: WHILE (got_operand) DO increment_lc;
      DS: lc = lc + operand_value;
      1_byte_instruction: lc = lc + 1;
      2_byte_instruction: lc = lc + 2;
      3_byte_instruction: lc = lc + 3;
    END
  END
  /***** PASS 2: CREATE THE OBJECT PROGRAM *****/
  rewind_input_file_pointer;
  lc = 0;
  mnemonic = null;
  open_output_file;
  WHILE (mnemonic != end) DO BEGIN
    get_line();
    scan_line();  /* determine mnemonic op code & value(s) of operands */
    /* Note: Symbols used in operand field are looked-up in the symbol */
    /* table created during pass one.                                  */
    CASE mnemonic OF BEGIN
      null, comment, EQU, END: ; /* do nothing */
      ORG: lc = operand_value;
      DB: WHILE (operand) BEGIN
          put_in_object_file(operand);
          lc = lc + 1;
        END
      DS: lc = lc + operand;
      1_byte_instruction:     put_in_object_file(inst_code);
      2_byte_instruction:     put_in_object_file(inst_code);
                              put_in_object_file(operand);
      1_byte_instruction:     put_in_object_file(inst_code);
                              put_in_object_file(operand high-byte);
                              put_in_object_file(operand low-byte);
      lc = lc + size_of_instruction;
    END
  END
  close_input_file;
  close_output_file;
END
```


图7-2 两次扫描的伪码概述

7.3 汇编语言程序格式

汇编语言程序包括：

- ❑ 机器指令
- ❑ 汇编器伪指令
- ❑ 汇编器控制
- ❑ 注释

机器指令即常见的可执行指令的助记符（如ANL）。汇编器伪指令是汇编器程序的指令，定义程序结构、符号、数据、常数等（如ORG）。汇编器控制用来设置汇编器模式和工作流程（如$TITLE）。注释用来说明指令序列的目的和功能，增强程序的可读性。

每行语句包含的机器指令或汇编器伪指令必须按照汇编器所能理解的规范格式编写，由空格或制表符将每行语句分割成不同的“字段”，汇编程序的一般语句格式为：

```
[label:] mnemonic [operand] [,operand] [.. .] [;comment]
```

其中只有助记符字段是必需的。一些汇编器要求：如果语句中存在标号字段，那么必须始于第1列的最左边，且随后的字段需要用空格或制表符间隔开。但是，对于ASM51，标号字段不需要从第1列开始，而且助记符字段可以和标号字段不在同一行上。但操作数字段必须和相应的助记符字段位于同一行内。各个字段详细描述如下。

7.3.1 标号字段

标号代表了紧随其后的指令或数据的地址。在需要分支转移到该指令时，相应的标号将作为操作数字段出现在分支和跳转指令（如SJMP SKIP）中。

“标号”总是代表地址，应该说“符号”是更具普遍意义的术语。标号属于符号的一种，其主要标志是必须以冒号作为结尾。通过EQU、SEGMENT、BIT、DATA等伪指令，可以给符号赋值或定义属性。符号可以是地址、数据常数、段名或编程者自定义结构名称等。符号不需要用冒号作结尾。在下面的例子中，PAR是符号而START是标号（符号的一种类型）。

```
PAR      EQU   500           ;"PAR" IS A SYMBOL WHICH
                             ;REPRESENTS THE VALUE 500
START:   MOV   A,#0FFH       ;"START" IS A LABEL WHICH
                             ;REPRESENTS THE ADDRESS OF
                             ;THE MOV INSTRUCTION
```

符号（或标号）必须以字母、问号或下划线开始，其后可以是字母、数字、“？”或下划线，长度不能超过31个字符①。符号可以使用大写或小写字母，汇编器

① 读者需要的限制是针对ASM51汇编器的，其他类型的汇编器可能有不同的要求。

不分辨大小写。预留字（助记符、运算符、预定义符号和伪指令）不能用作符号。

7.3.2 助记符字段

标号字段其后是助记符字段，可以是指令助记符，也可以是汇编器伪指令。指令助记符，如ADD、MOV、DIV或INC等；汇编器伪指令，如ORG、EQU或DB。汇编器伪指令将在本章的后续部分讨论。

7.3.3 操作数字段

操作数字段位于助记符字段后面。该字段包含指令需要用到的地址和数据。标号可用于代表地址，符号可用于代表数据常数。操作数字段的内容主要和所执行的操作相关，一些操作可能没有操作数（如RET指令），而另外一些指令可能需要多个操作数，这些操作数之间要用逗号隔开。事实上，操作数字段的内容多种多样，将会在后续部分中做详细的讨论，下面先看注释字段。

7.3.4 注释字段

注释字段位于每行的最后，用于阐明该行语句的执行功能等信息。注释必须以分号开始，如果1行语句以分号开始，那么该行为注释行。子例程和较大代码段的前面通常有一个注释块，也就是通过几个注释行来说明随后一段指令的性质。

156

7.3.5 特殊汇编器符号

特殊汇编器符号用于特定寄存器寻址模式，包括A、R0~R7、DPTR、PC、C和AB。另外，美元符号（$）代表定位计数器的当前值。下面是一些特殊汇编器符号的应用例子。

```
SETB C
INC  DPTR
JNB  TI,$
```

上面的最后1条指令有效利用了ASM51定位计数器，避免了标号的使用。该指令也可写成：

```
HERE:    JNB TI,HERE
```

7.3.6 间接寻址

对某些指令来说，操作数字段为一寄存器，其中存放着操作数的地址。符号@表示间接寻址，所采用的寄存器随指令的不同而异，但只能使用R0、R1、DPTR或者PC这几个寄存器。例如：

```
ADD    A,@R0
```

```
MOVC   A,@A+PC
```

上面的第1条指令，可以实现从R0寄存器内容所指定地址的内部RAM存储单元取得1字节的数据。第2条指令，将当前累加器A的内容加上程序计数器PC值，形成待访问的外部程序存储器的地址，CPU将从该地址取得1B的数据。需要注意的是，在通过加法求地址的时候，程序计数器PC的值是紧随MOVC指令的下一条指令的地址。执行完上述2条指令后，取回的操作数存入累加器A。

7.3.7 立即数

指令采用立即寻址方式提供成为指令一部分的操作数字段中的数据。立即数的标志是在其前面加有井号“#”，例如：

```
CONSTANT   EQU   100
           MOV   A,#0FEH
           ORL   40H,#CONSTANT
```

157

所有的立即数都要求是8位的数据（MOV DPTR，#data除外）。在表达式求值的时候，立即数被作为1个16位常数使用，但仅使用它的低位字节。高位字节的各个位必须相同（00H或FFH），否则，会产生“value will not fit in a byte”的错误信息提示。例如，下面2条指令在语法上是正确的。

```
MOV    A,#0FF00H
MOV    A,#00FFH
```

但下面2条指令会产生错误信息：

```
MOV    A,#0FE00H
MOV    A,#01FFH
```

如果用到了有符号的十进制常数（从–256到+255），例如，下面2条指令的执行效果是等效的，而且没有语法错误。

```
MOV    A,#-256
MOV    A,#0FF00H
```

2条指令的执行结果都是将00H传送到累加器A。

7.3.8 数据地址

对于一些采用直接寻址方式来访问内存的指令，需要在操作数字段给出相应的片上数据存储器的地址（00H~7FH）或者特殊功能寄存器的地址（80H~FFH），而且特殊功能寄存器的地址可以用预定义好的符号代替，例如：

```
MOV   A,45H
MOV   A,SBUF                  ;SAME AS MOV A,99H
```

7.3.9 位地址

8051最强大的特性之一是可以访问独立的位，而无需对字节进行掩码运算。当

需要通过指令访问内存空间的某个可位寻址的单元时，则要求提供该单元的位地址，包括内部RAM的128个位地址（00H~7FH），以及一些特殊功能寄存器的位地址（80H~0FFH）。

有3种方式可以用来指定指令中的位地址：（a）直接给出位地址；（b）将点运算符置于字节地址和相应的位在字节中的位置序号之间；（c）使用预先定义好的汇编器符号。下面分别给出相应的例子：

```
SETB   0E7H        ;EXPLICIT BIT ADDRESS
SETB   ACC.7       ;DOT OPERATOR (SAME AS ABOVE)
JNB    TI,$        ;"TI" IS A PRE-DEFINED SYMBOL
JNB    99H,$       ;(SAME AS ABOVE)
```

7.3.10 代码地址

代码地址一般用于跳转指令的操作数字段中，包括相对跳转（SJMP和条件跳转）、绝对跳转和调用（ACALL和AJMP）以及长跳转和调用（LJMP和LCALL）。

158

代码地址通常以标号的形式给出，例如：

```
HERE:   .
        .
        .
        SJMP HERE
```

ASM51在汇编时确定正确代码地址，然后根据指令的类型，分别在指令的适当位置插入8位的有符号偏移量、11位的页地址或16位的长地址。

7.3.11 通用的跳转和调用

ASM51允许编程者使用通用的JMP和CALL助记符，JMP可用来替代SJMP、AJMP和LJMP；CALL可用于替代ACALL和LCALL。汇编器在将通用助记符转换成“实际”指令时遵循以下简单的规则。如果没有前向引用，而且目标地址和源地址的差小于−128，通用助记符转换为短指令（仅适用于JMP）。如果没有前向引用，而且紧随JMP和CALL的指令和目标地址位于同一2K存储页之内，JMP和CALL分别被转换成AJMP和ACALL。在上述两种条件都不满足的情况下，JMP和CALL将被分别转换成LJMP和LCALL。

这种编译器所做的自动转换不是一个较好的编程选择。例如，如果是前向引用，即使在如果将JMP转换成SJMP会使效率更高的情况下，汇编器仍然会将JMP转成LJMP。在图7-3的汇编指令序列中，一共用到了3条通用的跳转指令，第1条跳转指令（位于第3行）将被汇编成SJMP，原因是，其目的地址是与其相邻的前1条指令（即非前向引用），而且地址之差小于−128。第4行的ORG指令在标号START和第2条跳转指令之间产生了200个存储单元的地址增量，所以，第5行的JMP被汇编成AJMP，原因是目标地址和源地址之间的偏移量超过了128。需要注意的是，该程序

159

所在的存储页面的地址范围为1000H~17FFH，第2条跳转指令的下一条指令的地址为12FEH，目标地址为1234H，都在此页面之内，而这也是JMP转成AJMP的必要条件之一。对于第3条JMP 指令，由于汇编到该指令时，目的标号FINISH还没有定义（前向引用），因而该JMP指令被转为LJMP。读者若是在目标文件中的相关位置检查每条跳转指令的机器码，就会发现汇编器对JMP所做的转换和上面的分析完全一致。可以参考附录C中SJMP、AJMP和LJMP的格式来校验目标文件中的十六进制代码。

```
LOC   OBJ      LINE   SOURCE
1234              1             ORG    1234H
1234 04           2    START:   INC    A
1235 80FD         3             JMP    START          ;ASSEMBLES AS SJMP
12FC              4             ORG    START + 200
12FC 4134         5             JMP    START          ;ASSEMBLES AS AJMP
12FE 021301       6             JMP    FINISH         ;ASSEMBLES AS LJMP
1301 04           7    FINISH:  INC    A
                  8             END
```

图7-3 通用JMP助记符的使用

7.4 汇编时的表达式求值

位于操作数字段的数据和常数有三种表述格式：(a) 显式定义（如0EFH）；(b) 采用预定义符号（如ACC）；(c) 表达式［如（2+3）］。在8051的汇编程序中采用表达式，增强了汇编语言程序的可读性和程序设计上的灵活性。当程序中使用了表达式时，汇编器将会对该表达式求值并插入到指令中的相应位置。

所有的表达式求值都按16位的算法来进行。但是根据需要，插入指令的数据既可能是8位的，也可能是16位的。下面两条指令是等效的：

```
MOV    DPTR,#04FFH + 3
MOV    DPTR,#0502H              ;ENTIRE 16-BIT RESULT USED
```

如果在指令MOV A，#data中应用和上面相同的表达式，ASM51汇编器在汇编时会产生“value will not fit in a byte”的错误提示消息。以下是表达式求值规则。

7.4.1 数基

8051按照Intel微处理器的通常做法来注明数字常数的基。常数结尾处的B表示二进制，O或Q表示八进制，D或者结尾处什么也没有表示十进制，H表示十六进制。例如，下面的指令是等价的：

```
MOV    A, #15
MOV    A, #1111B
MOV    A, #0FH
MOV    A, #17Q
```

```
MOV   A, #15D
```

注意，为了区别于标号，十六进制常数的第一个字母必须是数字（例如，是0A5H，而不是A5H）。

7.4.2 字符串

字符串由1个或2个字符构成，可用在操作数字段的表达式中，汇编器在汇编时，把字符串的ASCII码转换为等价的二进制形式。字符常数需要用单引号（'）括起来。请看下面的例子：

```
CJNEA,#'Q',AGAIN
SUBB A,#'0'            ;CONVERT ASCII DIGIT TO
                       ;BINARY DIGIT
MOV  DPTR,#'AB'
MOV  DPTR,#4142H       ;SAME AS ABOVE
```

160

7.4.3 算术运算

算术运算符包括：

```
+      加法
-      减法
*      乘法
/      除法
MOD    求模（即除法的余数）
```

例如，下面两条指令是等效的：

```
MOV  A, 10+10H
MOV  A, #1AH
```

下面两条指令也是等效的：

```
MOV  A, #25 MOD 7
MOV  A, #4
```

由于MOD运算符可能会和符号相混淆，因而在使用时，必须用至少1个空格或制表符号将MOD同其运算数分开，或者用括弧把参与运算的数用括号括起来。对于其他由字母构成的运算符，按照同样规则来处理。

7.4.4 逻辑运算符

逻辑运算符包括：

```
OR    逻辑或
AND   逻辑与
XOR   逻辑异或
NOT   逻辑非（补）
```

逻辑运算作用于操作数的对应位上。必须用空格或制表符号将逻辑运算符同其操作数分开。例如，下面两条指令等价：

```
MOV     A, #'9' AND 0FH
MOV     A, #'9
```

NOT运算符仅针对一个操作数，下面3条MOV指令是等效的：

```
THREE               EQU     3
MINUS_THREE         EQU     -3
                    MOV     A, #(NOT THREE) +1
                    MOV     A, #MINUS_THREE
                    MOV     A, #11111101B
```

7.4.5 特殊运算符

特殊运算符有：

```
SHR         右移
SHL         左移
HIGH        取高字节
LOW         取低字节
()          优先求值
```

例如，下面2条指令是等效的：

```
MOV         A, #8 SHL 1
MOV         A, #10H
```

下面两条指令也等价：

```
MOV         A, #HIGH 1234H
MOV         A, #12H
```

7.4.6 关系运算符

关系运算符的运算结果只有两种：假（0000H）和真（FFFFH）。关系运算符有：

```
EQ      =       等于
NE      <>      不等于
LT      <       小于
LE      <=      小于等于
GT      >       大于
GE      >=      大于等于
```

两种运算符形式都可以接受（例如“EQ”和“=”均可）。下面的例子中，所有的关系运算的结果都是“真”：

```
MOV     A, #5 = 5
MOV     A, #5 NE 4
MOV     A, #'X' LT 'Z'
MOV     A, #'X' >= 'X'
MOV     A, #$ > 0
MOV     A, #100 GE 50
```

所以上面所有的指令都等效于：

```
MOV    A, #0FFH
```

虽然表达式求值的结果有16位（例如0FFFFH），但在上面的例子中，MOV指令只能传送1个字节的数据，因此只用到了16位数据的低8位。在本例中，这一点对结果没有影响，因为对有符号的16位数据FFFFH和8位数据FFH来说都等于十进制的−1。

162

7.4.7 表达式例子

下面是一些表达式的例子以及相应的求值结果：

表 达 式	结 果	表 达 式	结 果
'B' - 'A'	0001H	'A' SHL 8	4100H
8/3	0002H	LOW 65535	00FFH
155 MOD 2	0001H	(8+1) * 2	0012H
4 * 4	0010H	5 EQ 4	0000H
8 AND 7	0000H	'A' LT 'B'	FFFFH
NOT 1	FFFEH	3 <=3	FFFFH

下面是一个初始化定时器的实例：将−500赋值到汇编器将−500转换为相应的16位数（FE0CH），HIGH和LOW运算符分别抽取高字节（FEH）和低字节（0CH）。

```
VALUE     EQU    -500
          MOV    TH1, #HIGH VALUE
          MOV    TL1, #LOW VALUE
```

汇编器将−500转换为相应的16位值（FE0CH），然后HIGH和LOW操作符分别提取出相应的高字节（FEH）和低字节（0CH），以此和每个MOV指令相呼应。

7.4.8 运算符优先级

表达式中的运算符的优先级从高到低排列如下：

```
()
HIGH LOW
*/ MOD SHL SHR
+-
EQ NE LT LE GT GE = <> < <= > >=
NOT
AND
OR  XOR
```

同级别的运算符，从左至右依次计算，举例如下：

表 达 式	值	表 达 式	值
HIGH ('A' SHL 8)	0041H	NOT 'A' -1	FFBFH
HIGH 'A'SHL 8	0000H	'A' OR 'A' SHL 8	4141H

163

7.5 汇编器指令

汇编器指令指导汇编器汇编程序，这些指令不是汇编语言的机器指令，也不能在目标微处理器上执行。但是指令在程序里处于助记符字段的位置。除了DB和DW以外，所有指令都不会影响程序存储器的内容。

ASM51可提供如下几类指令：

- ❑ 汇编器状态控制指令（ORG，END，USING）
- ❑ 符号定义指令（SEGMENT，EQU，SET，DATA，IDATA，XDATA，BIT，CODE）
- ❑ 存储空间初始化/预留指令（DS，DBIT，DB，DW）
- ❑ 程序链接指令（PUBLIC，EXTRN，NAME）
- ❑ 段选择指令（RSEG，CSEG，DSEG，ISEG，ESEG，XSEG）

下面按类别对汇编器指令进行介绍。

7.5.1 汇编器状态控制指令

1. ORG（set origin）指令

ORG（set origin）指令的格式如下：

```
ORG expression
```

ORG指令可以设置定位计数器的数值，来设定紧随其后的汇编语句的起始地址。ORG前面不允许使用标号。请看下面的例子：

```
ORG    100H                    ;SET LOCATION COUNTER TO 100H
ORG    ($ + 1000H) AND 0F000H  ;SET TO NEXT 4K BOUNDARY
```

在任何类型的段的内部都可以使用ORG指令，如果当前段使用的是绝对地址，那么ORG表达式的值代表当前段的一个绝对地址。如果ORG指令处在一个可重定位的段内，ORG表达式的值被看作相对于当前段基地址的偏移值。

2. END指令

END指令的格式如下：

```
END
```

END应当是源文件的最后一条语句，标示着汇编程序的结束。END前面不允许使用标号，而且汇编器不会汇编END后面的语句。

3. USING指令

USING指令的格式：

```
USING      expression
```

该指令通知ASM51汇编器当前可以使用的寄存器组（处于激活状态）。之后的语句中如果使用了AR0~AR7等预定义符号，汇编器会根据当前的工作寄存器组，把这些预定义符号转换为相应的直接地址。请看下面的指令：

```
USING       3
PUSH        AR7
USING       1
PUSH        AR7
```

第1条PUSH指令被转换为PUSH 1FH（寄存器组3中的R7寄存器），但第2条指令被转换为PUSH 0FH（寄存器组1中的R7寄存器）。

需要注意的是，USING指令并没有真正地切换寄存器组，该指令仅仅是通知汇编器，当前处于激活状态的寄存器组是哪个。对于寄存器组切换功能，8051有专门的指令负责。请看下面的例子，该例子对上面的例子做了部分修改。

```
MOV     PSW,#00011000B      ;SELECT REGISTER BANK 3
USING   3
PUSH    AR7                 ;ASSEMBLE TO PUSH 1FH
MOV     PSW,#00001000B      ;SELECT REGISTER BANK 1
USING   1
PUSH    AR7                 ;ASSEMBLE TO PUSH OFH
```

7.5.2 符号定义指令

符号定义指令定义代表段、寄存器、数字以及地址的符号。在这类指令前不能有标号。除了SET指令，采用该类指令定义的符号，既不能是前面已经定义过的符号，也不允许在后面语句中重新定义。

1. SEGMENT指令

SEGMENT指令的格式如下：

```
symbol     SEGMENT   segment_type
```

symbol是可重新定位段的名字。传统的汇编器只支持code和data等类型的段，相比之下，ASM51所支持的段类型较多。8051有多种类型的内存空间，为了利用这些空间，ASM51在code和data之外，还另外定义了多种段类型。下面是8051定义的各种段（内存空间）类型：

- CODE（代码段）
- XDATA（外部数据空间）
- DATA（可直接存取的内部数据空间，地址为00H~7FH）
- IDATA（间接存取的内部数据空间，地址为00H~7FH，在8052上是00H~FFH）
- BIT（位空间，对应内部数据空间字节地址为20H~2FH的存储单元）

例如，下面的语句：

```
EPROM      SEGMENT     CODE
```

定义符号EPROM代表一个CODE类型的段。请注意，该语句仅声明了EPROM的类型，如果要实际使用该段，必须用RSEG指令来给出更详细的定义（见下文）。

2. EQU（Equate）指令

EQU指令的格式如下：

```
symbol    EQU    expression
```

EQU指令为符号常数symbol分配1个数值，即表达式expression的计算值。表达式必须符合前文所提到的相应规则。下面是应用EQU的例子：

```
N27        EQU 27                ;SET N27 TO THE VALUE 27
HERE       EQU $                 ;SET "HERE" TO THE VALUE
                                 ;OF THE LOCATION COUNTER
CR         EQU 0DH               ;SET CR (CARRIAGE RETURN) TO 0DH
MESSAGE:   DB 'This is a message'
LENGTH     EQU $ - MESSAGE       ;"LENGTH" EQUALS LENGTH OF
                                  "MESSAGE"
```

3. 其他符号定义指令

SET指令的功能类似于EQU指令，但有一点区别：源程序中采用SET定义的符号，允许用另外1条SET予以重新定义。

DATA、IDATA、XDATA、BIT、CODE等指令，可以用来给相应段的地址赋予一个符号，这些指令并不是必不可少的。采用EQU指令也可以实现它们的功能。然而，如果用这些指令来定义符号，ASM51汇编器在汇编源代码的时候，会启动汇编器自带的功能强大的类型检查功能对源文件进行处理。考虑如下的两种指令和其他4条语句：

```
FLAG1        EQU    05H
FLAG2        BIT    05H
             SETB   FLAG1
             SETB   FLAG2
             MOV    FLAG1, #0
             MOV    FLAG2, #0
```

ASM51汇编器在汇编最后一条指令中的FLAG2的时候，会出现错误提示信息“data segment address expected”。由于FLAG2是采用BIT指令定义的位地址（采用BIT指令），因而可以用在SETB指令中，但不能用在MOV指令中（MOV指令要求操作数是字节），如果用了，编译器就会报错。在上述指令中，符号FLAG1的值和FLAG2相同，都是05H，但FLAG1是采用EQU指令定义的，没有和特定的存储器空间关联起来，因而不会报错。EQU指令对存储器地址空间的类型不敏感，与其说是优点，还不如说是缺点。程序设计者如果正确地定义符号，并把新定义的符号和特定的存储器空间联系起来（采用DATA、IDATA、XDATA等指令），那么，在汇编的时候，编程者会体会到ASM51汇编器的类型检查功能所带来的好处，从而可以避免出现误用符号的错误。

7.5.3 存储空间初始化/预留伪指令

采用存储空间初始化/预留伪指令可以实现初始化或者预留部分存储空间，单位

可以是字、字节或位。所预留的存储器空间的起始地址由当前段中定位计数器的数值决定。在这类指令之前允许附加标号。下面介绍存储空间的初始化/预留伪指令。

166

1. DS

DS（Define Storage）（定义存储空间）指令的格式如下：

```
[label:]    DS    expression
```

DS指令以字节为单位保留内存空间。除了BIT类型的段外，DS可用于所有的其他类型段中。要求表达式必须能够在汇编的时候求值，不能有前向引用、可重新定位的引用以及外部引用。如果程序中出现了DS语句，在汇编时，定位计数器的值将会被更新为其当前值和由表达式求值结果相加的和。但该求和结果不能超出当前地址空间的边界。

在下面的例子中，相关指令在内部数据段中生成了一个40B的缓冲区。

```
                DSEG AT 30H  ;PUT IN DATA SEGMENT (ABSOLUTE, INTERNAL)
LENGTH   EQU    40
BUFFER:  DS     LENGTH       ;40 BYTES RESERVED
```

标号BUFFER代表被保留空间的起始地址。在这个例子中，起始地址为30H（起始地址由DSEG指令定义，请参考7.5.5节）。下面的指令把该保留空间清零：

```
            MOV   R7, #LENGTH
            MOV   R0, #BUFFER
LOOP:       MOV   @R0, #0
            DJNZ  R7, LOOP
            (continue)
```

下面的指令在外部RAM中生成1000B的缓冲区，起始地址为4000H:

```
XSTART      EQU      4000H
XLENGTH     EQU      1000
            XSEG         AT         XSTART
XBUFFER:    DS XLENGTH
```

下面的指令把该缓冲区空间清零：

```
            MOV     DPTR, #XBUFFER
LOOP:       CLR     A
            MOVX    @DPTR, A
            INC     DPTR
            MOV     A, DPL
            CJNE    A, #LOW(XBUFFER+XLENGTH+1), LOOP
            MOV     A, DPH
            CJNE    A, #HIGH(XBUFFER+XLENGTH+1), LOOP
            (continue)
```

上面的指令序列是一个很好的关于ASM51运算符和汇编时表达式求值的例子。由于8051中没有指令能够完成将DPTR和立即数直接进行比较，因而必须灵活组合现有的指令，才能实现该功能。本例中进行了两次比较，分别对应DPTR的低字节和高

167

字节。由于CJNE（Compare-and-Jump-if-Not-Equal）指令的操作数只能是累加器或寄存器，因而在比较前，必须把DPTR的相应字节传送到累加器。当DPTR的值等于XBUFFER+XLENGTH+1的时候，LOOP循环终止。［表达式中的"+1"是必需的，因为8051在最后一次执行MOVX指令后（即清零最后一个存储单元），还要将DPTR的值增加1；换言之，若去掉"+1"，其结果是所定义的缓冲区空间的最后一个单元没有被清零。］

2. DBIT指令

DBIT指令的格式为：

```
[label:]    DBIT expression
```

DBIT指令以位为单位来保留存储空间，只能用于BIT类型的段内。表达式必须能够在汇编的时候求值，且不能有前向引用。如果在程序中用到了DBIT语句，那么汇编器在扫描源程序时先对表达式求值，然后把当前段（BIT类型的段）的定位计数器的值和表达式的值相加，求和的结果作为定位计数器的新值。需要注意，在一个BIT类型的段中，定位计数器的基本单位是位而非字节。下面的指令序列在一个BIT类型的段中，生成了三个代表存储空间中相应位的标号。

```
          BSEG              ;BIT SEGMENT (ABSOLUTE)
KBFLAG:   DBIT 1            ;KEYBOARD STATUS
PRFLAG:   DBIT 1            ;PRINTER STATUS
DKFLAG:   DBIT 1            ;DISK STATUS
```

上面的例子中，BSEG指令没有给出当前段的地址，DBIT指令定义了3个标号，它们对应的位地址可从汇编器生成的.LST文件或.M51文件的符号表中（请参考图7-1和图7-6）找到。如果是第一次用BSEG指令，那么，KBFLAG对应的位地址是00H（内部RAM中字节地址为20H存储单元的第0个位，请参考图2-6）。如果在前面语句中已经使用了BSEG指令，那么，KBFLAG的位地址紧接最后一个由DBIT定义的位地址（请参考7.5.5节）。

3. DB指令

DB（Define Byte）指令的格式：

```
[label:]    DB    expression [, expression][. . .]
```

DB指令以字节为单位初始化代码存储器空间。因为要在代码存储器空间中存放一些数据常数，所以必须先激活一个CODE类型的段。表达式列表可由1个或多个字节组成，每个字节都能以独立的表达式形式出现，各个表达式之间以逗号隔开。

DB指令允许表达式中有被单引号包起来的字符串，字符串的长度可以超过2字节（但不能超过表达式规定范围）。汇编器会把字符串中的每个字符转换为相应的ASCII码。如果使用了标号，汇编器会将字符串的第1个字节的地址作为标号对应的地址。请看如下语句：

```
          CSEG AT 0100H
SQUARES:  DB   0,1,4,9,16,25      ;SQUARES OF NUMBERS 0-5
MESSAGE:  DB   'Login:',0         ;NULL-TERMINATED CHARACTER STRING
```

汇编后，外部代码空间中的赋值情况如下（十六进制）：

地　址	内　容	地　址	内　容
0100	00	0107	6F
0101	01	0108	67
0102	04	0109	69
0103	09	010A	6E
0104	10	010B	3A
0105	19	010C	00
0106	4C		

4. DW指令

DW（Define Word）指令的格式：

```
[label:]   DW   expression      [,expression][. . .]
```

DW指令和DB指令的功能相同，但以双字节（16位）作为数据的存储单位。例如：

```
CSEG  AT  200H
DW    $,'A',1234H, 2, 'BC'
```

汇编后，外部代码空间中的赋值情况如下（十六进制）：

地　址	内　容	地　址	内　容
0200	02	0205	34
0201	00	0206	00
0202	00	0207	02
0203	41	0208	42
0204	12	0209	43

7.5.4 程序链接指令

程序链接指令可以给模块命名并允许模块间相互引用，从而使得几个已经单独汇编的模块（文件）可以协同工作。在下面的讨论中，模块可以被看作等同于文件（但事实上，1个模块可能会涉及多个文件）。

1. PUBLIC指令

PUBLIC指令的格式如下：

```
PUBLIC symbol [,symbol][. . .]
```

PUBLIC指令定义的符号（或符号列表），可以在当前模块以外访问并使用。1个被声明为 PUBLIC的符号必须在当前模块内定义，但该符号可以被其他模块所引用。例如：

```
PUBLIC INCHAR, OUTCHR, INLINE, OUTSTR
```

2. EXTRN指令

EXTRN指令的格式如下：

```
EXTRN segment_type(symbol [,symbol][. . .], . . .)
```

EXTRN指令列出那些在本模块内被引用到，却在另外的模块中定义的符号。所有列出的符号，必须和某个段类型关联起来（DATA、IDATA、XDATA、BIT、CODE以及NUMBER，NUMBER是那些由EQU定义且没有给出段类型的符号）。符号所关联的段类型规定了这些符号的使用方式。在汇编器链接不同的模块时，符号所关联的段类型可以帮助汇编器检查这些符号在不同模块中的引用是否正确。

PUBLIC指令和EXTRN指令协同工作，例如下面图7-4中的两个文件：MAIN.SRC和MESSAGE.SRC。子例程HELLO和GOOD_BYE定义在MESSAGE模块内，但被声明为PUBLIC，因而，其他模块可以引用这两个符号。在MAIN模块中，虽然没有定义HELLO和GOOD_BYE这两个子例程，但MAIN模块可以调用这两个子例程，因为已经用EXTRN指令声明这二者是定义在其他模块中的符号。

MAIN.SRC

```
            EXTRN    CODE(HELLO,GOOD_BYTE)
            ...
            CALL        HELLO
            ...
            CALL     GOOD_BYE
            ...
            END
```

MESSAGES.SRC

```
            PUBLIC      HELLO,GOOD_BYE
            ...
HELLO:      (begin subroutine)
            ...
            RET
GOOD_BYE:   (begin subroutine)
            ...
            RET
            ...
            END
```

图7-4 EXTRN和PUBLIC汇编指令的应用

MAIN.SRC和MESSAGE.SRC都不能独立构成一个完整的程序。二者单独汇编后必须链接起来才能组成一个可执行的程序。外部引用问题在链接时得到解决，汇编器会在CALL指令处插入正确的目标地址。

3. NAME指令

NAME指令的格式如下：

```
NAME module_name
```

模块命名的规则和符号相同。如果编程者没有给出一个模块的名字，那么该模块默认的名称和文件名相同（不包括盘符、路径名和文件的扩展名）。源程序文件中如果没有NAME指令，汇编器会认为整个文件就是一个模块。因而“模块”这个概态，对于那些规模相对较小的程序而言，其含义显得有点多余，就是对那些中等规模的程序（例如，源程序包括几个文件，而各个文件中有可重定位的段）而言，也没有必要使用NAME指令，也无需对“模块”的概念给予特别关注。因此，为了简化ASM51的学习，在模块的定义中提到，可以把模块看作一个“文件”。但如果程序非常大（源程序包括几千行甚至更多的代码），将程序划分为不同的模块就显得非常有意义了，此时，每个模块可能包含几个文件，其中包含若干用于共同目的的例程。

7.5.5 段选择指令

汇编器在汇编的时候，如果遇上段选择指令，会把紧随其后的代码或数据（直到遇上另外一个段选择指令为止）置于所选择的段内。该指令所选择的段，既可以是前面定义的可重定位的段，也可以是当即生成的、使用绝对地址的段。

1. RSEG伪指令（可重定位的段）

RSEG 伪指令的格式如下：

```
RSEG segment_name
```

其中segment_name是前面用SEGMENT伪指令定义的可重定位的段的名称。RSEG是一个“段选择”伪指令，负责把紧随其后的代码或数据（直到出现另外一个段选择伪指令为止）置入名为segment_name的段。

2. 选择使用绝对地址的段

RSEG指令选择的是可重定位的段。另一方面，如下的指令选择的都是使用“绝对地址”的段：

```
CSEG  [AT address]
DSEG  [AT address]
ISEG  [AT address]
BSEG  [AT address]
XSEG  [AT address]
```

171

上述指令分别指定位于程序存储器、内部数据存储器、间接寻址的内部数据存储器、可位寻址区以及外部数据存储空间中的使用绝对地址的段。如果提供了绝对地址（通过AT address方式给出），汇编器会结束上一个使用绝对地址的同类型的段，以该地址为起始地址新建立一个相应类型且使用绝对地址的段。如果没有用AT给出起始地址，汇编器会选择前面最后选定的同类型使用绝对地址的段，并保持地址的连续性。如果汇编器在前面没有选定该类型的段，又没有给出绝对地址，那么，汇编器会新生成1个使用绝对地址的段，该段的起始地址为0。请注意，段中不能包括前

向引用，起始地址必须为绝对地址。

每个段都有属于自己的定位计数器，初始值都设为0。默认的段类型是使用绝对地址的代码段。因此，汇编器的初始状态为使用绝对地址的代码段的0000H单元。当其他段被第一次选择时，前一个被选择段所属定位计数器的最后的值被保存起来，当汇编器重新选择该段时，定位计数器从上一次最后保存的值开始重新计数。使用ORG指令可在当前选择的段内，改变当前段的定位计数器的值。下面的例子展示了如何定义和初始化可重定位的段以及使用绝对地址的段。

在图7-5中，最开始两行代码声明了符号ONCHIP和EPROM分别代表DATA类型（内部数据）和CODE类型（代码）的段。第4行定义了一个采用绝对地址的位数据段，起始的位地址是70H（内部RAM中地址为2EH字节的第0个位，请参考图2-6）。接下去，定义了两个标号：FLAG0和FLAG1，位地址是70H和71H。第8行的RSEG伪指令定义一个可重定位的内部数据段，段名为ONCHIP。标号TOTAL和COUNT对应着单字节空间的地址，标号SUM16对应着一个双字节空间的地址。第13行重新出现的RSEG指令，定义了一个可重定位的代码段，段名为EPROM。标号BEGIN代表了EPROM第一条指令的地址。请注意，单从图7-5的代码中，无法确定TOTAL、COUNT、SUM16、BEGIN等标号的绝对地址，原因是这些标号位于可重定位的段内。在把代码汇编成目标文件后，必须先指定ONCHIP段和EPROM段的起始地址，经由链接器/定位器处理才能确定这些标号所对应的地址。链接器/定位器在处理了目标文件后，会生成后缀名为.M51的列表文件，其中给出了标号的绝对地址。至于FLAG1标号和FLAG2标号，因为定义在使用绝对地址的位数据段，所以总是对应着位地址70H和71H。

172

```
LOC   OBJ           LINE    SOURCE

                      1     ONCHIP  SEGMENT DATA      ;relocatable data segment
                      2     EPROM   SEGMENT CODE      ;relocatable code segment
                      3
----                  4             BSEG    AT 70H    ;begin absolute bit segment
0070                  5     FLAG1:  DBIT    1
0071                  6     FLAG2:  DBIT    2
                      7
----                  8             RSEG    ONCHIP    ;begin relocatable data segment
0000                  9     TOTAL:  DS      1
0001                 10     COUNT:  DS      1
0002                 11     SUM16:  DS      2
                     12
----                 13             RSEG    EPROM     ;begin relocatable code segment
0000 750000     F    14     BEGIN:  MOV     TOTAL,#0
                     15             (continue program)
                     16             END
```

图7-5 绝对地址和可重定位段的定义及初始化

7.6 汇编器控制项

汇编器控制项用来调控ASM51汇编器的行为，使其按照一定的格式生成列表文件和目标文件。在很大程度上，汇编器控制项对程序自身没有丝毫的影响，但会影响列表文件的格式。在汇编某个程序时，控制项可由命令行上输入，也可以置于源程序中，如果是这样，必须要有美元$符号作为前缀，且必须位于第一列。

存在两类汇编器控制项，基本的（primary）和通用的（general）。基本控制项可从命令行上输入，也可置于源程序的起始处。在一个基本控制项前，只能是其他的基本控制项。通用控制项可位于源程序的任何位置。图7-6列出了ASM51所支持的汇编器控制项。

控制项名	基本的(P)/通用的(G)	默认值	缩写	含义
DATE(date)	P	DATE( )	DA	将字符串置于列表文件的文件头（最多9个字符）
DEBUG	P	NODEBUG	DB	输出相应调试符号信息到目标文件中
NODEBUG	P	NODEBUG	NODB	目标文件不包含相应调试符号信息
EJECT	G	不适用	EJ	在下一页继续列出
ERRORPRINT(file)	P	NOERRORPRINT	EP	除了列表文件,指定一个接受错误信息的文件(默认是控制台)
NOERRORPRINT	P	NOERRORPRINT	NOEP	指定只有列表文件能接受错误信息
GEN	G	GENONLY	GO	展开宏调用,列出所有宏展开后的源代码
GENONLY	G	GENONLY	NOGE	仅仅列出初始的源代码
INCLUDE(file)	G	不适用	IC	指定一个可嵌入于初始源程序文件的文件
LIST	G	LIST	LI	在列表文件中,列出LIST控制项后的源代码
NOLIST	G	LIST	NOLI	在列表文件中,不列出NOUST控制项后的源代码
MACRO(mem_percent)	P	MACRO(50)	MR	求值并展开所有的宏调用,指定可用内存中,用于宏处理的内存占的百分比
NOMACRO	P	MACRO(50)	NOMR	不对宏调用进行求值
MOD51	P	MOD51	MO	汇编器能够识别8051所专用的、预定义的特殊功能寄存
NOMOD51	P	MOD51	NOMO	汇编器不支持8051所专用的、预定义的特殊功能寄存器

图7-6 ASM51所支持的汇编控制项

控制项名	基本的(P)/通用的(G)	默认值	缩写	含义
OBJECT(file)	P	OBJECT(source.OBJ)	OJ	指定目标代码文件名
NOOBJECT	P	OBJECT(source.OBJ)	NOOJ	不生成目标代码文件
PAGING	P	PAGING	PI	列表文件分页,每页带有一个页头
NOPAGING	P	PAGING	NOPI	列表文件中不包含分页中断标记
PAGELENGTH(N)	P	PAGELENGT(60)	PL	指定列表文件分页后,每页的最大行数（10~65 536）
PAGEWIDTH(N)	P	PAGE WIDTH((120)	PW	指定列表文件每行的最大字符数(72~132)
PRINT(file)	P	PRINT(source.LST)	PR	指定列表文件的名称
NOPRINT	P	PRINT(source.LST)	NOPR	不生成列表文件
SAVE	G	不适用	SA	将当前控制项设置保存到SAVE栈
RESTORE	G	不适用	RS	从SAVE栈恢复控制项设置
REGISTERBANK(rb,...)	P	REGISTERBNAK(0)	RB	标明程序中使用了寄存器组
NOREGISTERBANK	P	REGISTERBNAK(0)	NORB	表明程序中没有使用寄存器组
SYMBOLS	P	SYMBOLS	SB	为程序中所有的符号,生成一张格式化表
NOSYMBOLS	P	SYMBOLS	NOSB	不生成符号表
TITLE(string)	G	TITLE()	TT	在接下来的每页的页头添加一字符串(最长60个字符)
WORKFILES(path)	P	同源文件一致	WF	指定临时工作文件的替代文件
XREF	P	NOXREF	XR	生成一个程序中所有符号的交叉引用列表文件
NOXREF	P	NOXREF	NOXR	不会生成交叉引用列表文件

173 ~ 175

图7-6 （续）

7.7 链接操作

在大型应用程序的开发过程中，常常把任务分解成多个子例程（模块），每个模块包含可以独立编写的代码（通常是子例程），这就是通常讲的“模块化编程”策略。一般情况下，各个模块是可重定位的，并不是固定在程序或数据空间的某一部分，因而，需要链接和定位程序，将各个模块整合成1个使用绝对地址的可执行目标模块。

Intel公司的RL51是一个典型的链接器/定位器。RL51读入若干可重定位的目标模块，生成1个可执行的机器语言程序（PROGRAM），同时生成1个列表文件(PROGRAM.M51)，该文件包含程序的内存布局和符号表，如图7-7所示。

在整合可重定位的模块时，汇编器会把各个模块所包含的外部符号的值确定下来，并写入最终的输出文件中。在系统提示符下，输入下列命令，会启动链接器：

```
RL51 input_list [TO output_file] [location_controls]
```

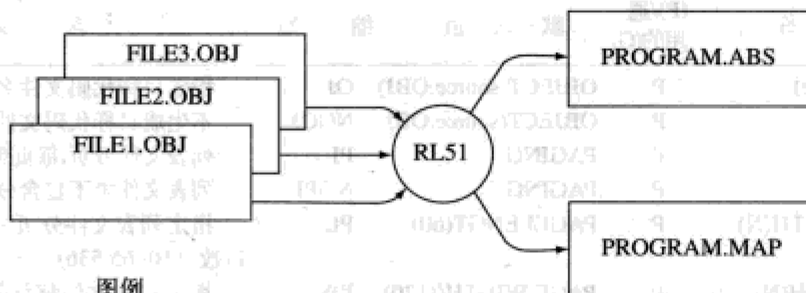


图7-7 链接器工作流程

input_list是可重定位的目标模块（文件）的列表，模块之间用逗号隔开。output_file是最后生成的采用绝对地址的目标模块的名称。如果未指定名称，生成文件的名称默认和第一个输入文件名相同，但没有后缀。定位控制项（location_control）用于设置已命名段的起始地址。

例如，假设要将3个模块或文件（MAIN.OBJ、MESSAGES.OBJ和SUBROUTINES.OBJ）整合成可执行程序（EXAMPLE），其中每个模块都带有两个可重定位的段，一个是EPROM，类型为CODE，另外一个是ONCHIP，类型为DATA。再假设，代码段的起始地址是4000H，数据段的起始地址是30H（内部RAM）。链接器调用如下：

```
RL51 MAIN.OBJ,MESSAGES.OBJ, SUBROUTINES.OBJ TO EXAMPLE &
CODE(EPROM(4000H))   DATA(ONCHIP(30H))
```

其中，&符号用作续行符。

如果程序的起始处是标号START，而START标号后面是MAIN模块中的第一条指令，那么程序从地址4000H处开始执行。如果MAIN模块不是第一个被链接的模块，或者标号START不在MAIN模块的开始处，那么检查由RL51生成的列表文件EXAMPLE.M51，可从它的符号表中获知程序的入口。默认情况下，EXAMPLE.M51仅仅包含链接的映射表。若要使其包含符号表，那么必须在每个源程序模块中使用SDEBUG控制项（请参考图7-6）。

7.8 例子详解——链接可重定位的段和模块

本节给出一个简单的8051程序，该例子带有详细的注解，包括很多前面刚介绍的概念。源代码由两个文件组成，在文件中使用EXTRN或PUBLIC指令声明了若干符号，这些符号用于文件之间的通信。1个文件就是1个模块，其中1个模块名称是MAIN，另外1个模块名称是SUBROUTINES。源程序中定义了1个可重新定位的代

码段EPROM，另外还定义了一个可重新定位的内部数据段ONCHIP。在程序规模很大的场合，将程序划分成多个文件、模块和段，这是很有效的编程策略。仔细阅读例程可以加深对上述概念的理解，对读者准备着手设计基于8051的实际系统也大有益处。

例子很简单，主要功能就是利用8051的串行端口、VDT键盘以及CRT显示器进行输入和输出操作，涉及如下内容：

- ❑ 初始化串行端口；
- ❑ 在显示器上显示提示符“Enter a command:”；
- ❑ 从键盘读取一行输入，读取每个字符时，在显示器上显示读入的字符；
- ❑ 将输入的一行字符串回显；
- ❑ 重复上述步骤。

图7-8的（a）是第1个源程序文件的列表文件（ECHO.LIST）；（b）是第2个源程序文件的列表文件（IO.LIST）；（c）是由链接器/定位器生成的列表文件(EXAMPLE.M51)。

```
MCS-51 MACRO ASSEMBLER    *** ANNOTATED EXAMPLE (MAIN MODULE) ***          03/17/91

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN ECHO.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE ECHO.SRC EP

LOC  OBJ            LINE     SOURCE

                       1     $DEBUG
                       2     $TITLE(*** ANNOTATED EXAMPLE (MAIN MODULE) ***)
                       3     $PAGEWIDTH(98)
                       4     $NOPAGING
                       5
                       6             NAME        MAIN               ;MODULE NAME IS "MAIN"
                       7             EXTRN CODE(INIT,OUTSTR)        ;DECLARE EXTERNAL SYMBOLS
                       8             EXTRN CODE(INLINE,OUTLINE)
                       9
 0000                 10     CR      EQU         0DH                ;CARRIAGE RETURN CODE
                      11     EPROM   SEGMENT     CODE               ;DEFINE SYMBOL "EPROM"
                      12
----                  13             RSEG        EPROM              ;BEGIN CODE SEGMENT
0000 120000     F     14     MAIN:   CALL        INIT               ;INITIALIZE SERIAL PORT
0003 900000     F     15     LOOP:   MOV         DPTR,#PROMPT       ;SEND PROMPT
0006 120000     F     16             CALL        OUTSTR
0009 120000     F     17             CALL        INLINE             ;GET A COMMAND LINE AND
000C 120000     F     18             CALL        OUTLINE            ; ECHO IT BACK
000F 80F2             19             JMP         LOOP               ;REPEAT
                      20
0011 0D               21     PROMPT: DB          CR,'Enter a command: ',0
0012 456E7465
0016 72206120
001A 636F6D6D
001E 616E643A
0022 20
0023 00
                      22             END
```

(a) ECHO.LST

图7-8　附有注释的例程：可重定位段和模块的链接

```
SYMBOL TABLE LISTING
------ ----- -------

N A M E    T Y P E  V A L U E    A T T R I B U T E S

CR . . . .    NUMB   000DH    A
EPROM. . .  C SEG    0024H             REL=UNIT
INIT . . .  C ADDR   ----        EXT
INLINE . .  C ADDR   ----        EXT
LOOP . . .  C ADDR   0003H    R        SEG=EPROM
MAIN . . .  C ADDR   0000H    R        SEG=EPROM
OUTLINE. .  C ADDR   ----        EXT
OUTSTR . .  C ADDR   ----        EXT
PROMPT . .  C ADDR   0011H    R        SEG=EPROM

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

(a) ECHO.LST

```
MCS-51 MACRO ASSEMBLER    *** ANNOTATED EXAMPLE (SUBROUTINES MODULE) ***    03/17/91

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN IO.OBJ
ASSEMBLER INVOKED BY:  C:\ASM51\ASM51.EXE IO.SRC EP

LOC  OBJ          LINE     SOURCE

                     1     $DEBUG
                     2     $TITLE(*** ANNOTATED EXAMPLE (SUBROUTINES MODULE) ***)
                     3     $PAGEWIDTH(98)
                     4     $NOPAGING
                     5
                     6             NAME     SUBROUTINES      ;MODULE NAME
                     7             PUBLIC   INIT,OUTCHR,INCHAR ;DECLARE PUBLIC SYMBOLS
                     8             PUBLIC   INLINE,OUTLINE,OUTSTR
                     9
                    10     ;*******************************************************************
                    11     ; DEFINE SYMBOLS                                                    *
                    12     ;*******************************************************************
  000D              13     CR      EQU      0DH              ;CARRIAGE RETURN
  0028              14     LENGTH  EQU      40               ;40-CHARACTER BUFFER
                    15     EPROM   SEGMENT  CODE             ;"EPROM" IS A CODE SEGMENT
                    16     ONCHIP  SEGMENT  DATA             ;"ONCHIP" IS A DATA SEGMENT
                    17
----                18             RSEG     EPROM            ;BEGIN RELOCATABLE CODE SEGMENT
                    19
                    20     ;*******************************************************************
                    21     ; INITIALIZE THE SERIAL PORT                                        *
                    22     ;*******************************************************************
0000 759852         23     INIT:   MOV      SCON,#52H        ;8-BIT UART MODE
0003 758920         24             MOV      TMOD,#20H        ;TIMER 1 SUPPLIES BAUD RATE CLOCK
0006 758DF3         25             MOV      TH1,#-13         ;2400 BAUD
0009 D28E           26             SETB     TR1              ;START TIMER
000B 22             27             RET
                    28
                    29     ;*******************************************************************
                    30     ; OUTPUT CHARACTER IN ACC (NOTE: VDT MUST CONVERT CR INTO CR/LF)    *
                    31     ;*******************************************************************
000C 3099FD         32     OUTCHR:  JNB     TI,$             ;WAIT FOR TRANSMIT BUFFER EMPTY
000F C299           33             CLR      TI               ;WHEN EMPTY, CLEAR FLAG AND
0011 F599           34             MOV      SBUF,A           ; SEND CHARACTER
0013 22             35             RET
                    36
                    37     ;*******************************************************************
                    38     ; INPUT CHARACTER TO ACC                                            *
                    39     ;*******************************************************************
```

(b) IO.LST

图7-8 (续)

```
0014 3098FD          40   INCHAR:   JNB    RI,$        ;WAIT FOR RECEIVE BUFFER FULL
0017 C298            41             CLR    RI          ;WHEN CHAR ARRIVES, CLEAR FLAG &
0019 E599            42             MOV    A,SBUF      ; INPUT CHAR TO ACC
001B 22              43             RET
                     44
                     45   ;****************************************************************
                     46   ; OUTPUT NULL-TERMINATED STRING                                  *
                     47   ;****************************************************************
001C E4              48   OUTSTR:   CLR    A           ;DPTR POINTS TO STRING OF CHAR
001D 93              49             MOVC   A,@A+DPTR   ;GET CHARACTER
001E 6006            50             JZ     EXIT        ;IF NULL BYTE, DONE
0020 120000    F     51             CALL   OUTCHR      ;OTHERWISE, SEND IT
0023 A3              52             INC    DPTR        ;POINT TO NEXT CHARACTER
0024 80F6            53             JMP    OUTSTR      ; AND SEND IT TOO
0026 22              54   EXIT:     RET
                     55
                     56   ;****************************************************************
                     57   ; INPUT CHARACTERS TO BUFFER                                     *
                     58   ;****************************************************************
0027 7800      F     59   INLINE:   MOV    R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0029 120000    F     60   AGAIN:    CALL   INCHAR      ;GET A CHARACTER
002C 120000    F     61             CALL   OUTCHR      ; ECHO IT BACK
002F F6              62             MOV    @R0,A       ;PUT IT IN BUFFER
0030 08              63             INC    R0          ;INCREMENT POINTER TO BUFFER
0031 B40DF5          64             CJNE   A,#CR,AGAIN ;IF NOT CR, GET ANOTHER CHAR
0034 7600            65             MOV    @R0,#0      ;PUT NULL BYTE AT END
0036 22              66             RET
                     67
                     68   ;****************************************************************
                     69   ; OUTPUT CONTENTS OF BUFFER                                      *
                     70   ;****************************************************************
0037 7800      F     71   OUTLINE:  MOV    R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0039 E6              72   AGAIN2:   MOV    A,@R0       ;GET CHARACTER FROM BUFFER
003A 6006            73             JZ     EXIT2       ;IF NULL BYTE, DONE
003C 120000    F     74             CALL   OUTCHR      ;OTHERWISE, SEND IT
003F 08              75             INC    R0          ;POINT TO NEXT CHAR IN BUFFER
0040 80F7            76             JMP    AGAIN2      ; AND SEND IT TOO
0042 22              77   EXIT2:    RET
                     78
                     79   ;****************************************************************
                     80   ; CREATE A BUFFER IN ONCHIP RAM                                  *
                     81   ;****************************************************************
----                 82             RSEG   ONCHIP      ;BEGIN RELOCATABLE DATA SEGMENT
0000                 83   BUFFER:   DS     LENGTH      ;ALLOCATE INTERNAL RAM AS BUFFER
                     84             END
```

(b) IO.LST

```
SYMBOL TABLE LISTING
------ ----- -------

N A M E      T Y P E   V A L U E      A T T R I B U T E S

AGAIN . . .  C ADDR    0029H    R     SEG=EPROM
AGAIN2. . .  C ADDR    0039H    R     SEG=EPROM
BUFFER. . .  D ADDR    0000H    R     SEG=ONCHIP
CR. . . . .    NUMB    000DH    A
EPROM . . .  C SEG     0043H          REL=UNIT
EXIT. . . .  C ADDR    0026H    R     SEG=EPROM
EXIT2 . . .  C ADDR    0042H    R     SEG=EPROM
INCHAR. . .  C ADDR    0014H    R PUB SEG=EPROM
INIT. . . .  C ADDR    0000H    R PUB SEG=EPROM
INLINE. . .  C ADDR    0027H    R PUB SEG=EPROM
LENGTH. . .    NUMB    0028H    A
ONCHIP. . .  D SEG     0028H          REL=UNIT
OUTCHR. . .  C ADDR    000CH    R PUB SEG=EPROM
OUTLINE . .  C ADDR    0037H    R PUB SEG=EPROM
OUTSTR. . .  C ADDR    001CH    R PUB SEG=EPROM
RI. . . . .  B ADDR    0098H.0 A
```

(c) EXAMPLE.M51

图7-8 (续)

```
SBUF. . . .  D ADDR    0099H    A
SCON. . . .  D ADDR    0098H    A
SUBROUTINES  ----      -----
TH1 . . . .  D ADDR    008DH    A
TI. . . . .  B ADDR    0098H.1  A
TMOD. . . .  D ADDR    0089H    A
TR1 . . . .  B ADDR    0088H.6  A

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND

DATE : 03/17/91
DOS 3.31 (038-N) MCS-51 RELOCATOR AND LINKER V3.0, INVOKED BY:
C:\ASM51\RL51.EXE ECHO.OBJ,IO.OBJ TO EXAMPLE CODE(EPROM(8000H))DATA(ONCHIP(30H
>> ))

INPUT MODULES INCLUDED
  ECHO.OBJ(MAIN)
  IO.OBJ(SUBROUTINES)

LINK MAP FOR EXAMPLE(MAIN)

              TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
              ----    ----      ------    ----------   ------------

              REG     0000H     0008H                  "REG BANK 0"
                      0008H     0028H                  *** GAP ***
              DATA    0030H     0028H     UNIT         ONCHIP

                      0000H     8000H                  *** GAP ***
              CODE    8000H     0067H     UNIT         EPROM

SYMBOL TABLE FOR EXAMPLE(MAIN)

VALUE          TYPE          NAME
-----          ----          ----

-------        MODULE        MAIN
N:000DH        SYMBOL        CR
C:8000H        SEGMENT       EPROM
C:8003H        SYMBOL        LOOP
C:8000H        SYMBOL        MAIN
C:8011H        SYMBOL        PROMPT
-------        ENDMOD        MAIN

-------        MODULE        SUBROUTINES
C:804DH        SYMBOL        AGAIN
C:805DH        SYMBOL        AGAIN2
D:0030H        SYMBOL        BUFFER
N:000DH        SYMBOL        CR
C:8000H        SEGMENT       EPROM
C:804AH        SYMBOL        EXIT
C:8066H        SYMBOL        EXIT2
C:8038H        PUBLIC        INCHAR
C:8024H        PUBLIC        INIT
C:804BH        PUBLIC        INLINE
N:0028H        SYMBOL        LENGTH
D:0030H        SEGMENT       ONCHIP
C:8030H        PUBLIC        OUTCHR
C:805BH        PUBLIC        OUTLINE
C:8040H        PUBLIC        OUTSTR
B:0098H        SYMBOL        RI
D:0099H        SYMBOL        SBUF
D:0098H        SYMBOL        SCON
D:008DH        SYMBOL        TH1
B:0098H.1      SYMBOL        TI
D:0089H        SYMBOL        TMOD
B:0088H.6      SYMBOL        TR1
-------        ENDMOD        SUBROUTINES
```

(c) EXAMPLE.M51

图7-8 (续)

7.8.1 ECHO.LST

ASM51汇编源文件（ECHO.SRC）之后，会生成一个列表文件ECHO.LST，图7-8a即为该列表文件的内容。列表文件中的开始几行是有关编程环境的通用信息。此外，还包括启动ASM51汇编器执行此次汇编操作的命令行输入情况，在汇编器程序名（即ECHO.SRC）之前新添加了汇编器的所在路径。需要注意的是，命令行上包括汇编器控制项EP（ERRORPINT）。该控制项通知汇编器，在输出错误信息的时候，除了要输出到列表文件之中，还要把出错信息输出到控制台上（请参考图7-6）。

源代码文件显示在列表SOURCE的下面，其左边是行号（LINE）列。显而易见，ECHO.SRC包含22行源代码。1~4行是汇编器控制项。第1行的$DEBUG指示ASM汇编器在目标文件ECHO.BOJ中建立符号表。如果要进行硬件仿真或者在通过链接器/定位器生成相应列表文件的时候，必须要有这张符号表。$TITLE定义显示在各页列表文件顶部的字符串。$PAGEWIDTH规定了列表文件中每行的最大宽度。$NOPAGING要求汇编器不要在列表文件中插入页中断（换页）标记。大部分汇编器控制项都对列表文件的格式有影响。为了得到所需要的排版格式，通常需要重复试验和纠错的过程。

第6行的NAME指令声明当前文件属于MAIN模块。在本例中，MAIN模块不再包含其他的源程序文件，但如果项目很大，MAIN模块可能还包含若干其他文件，这些文件也要用NAME指令声明自身属于MAIN模块。由于本例中MAIN模块仅包含1个文件，所以读者可姑且认为模块和文件是等同的概念。

第7行和第8行的EXTRN指令声明了哪些在本模块中会用到，但定义在其他模块中的符号。假使没有用EXTRN指令说明这些符号，汇编器处理源代码时，如果发现某行语句用到了这些符号，就会在该行提示“undefined symbol”的出错信息。为了确保能正确地使用外部符号，程序编写者必须定义每个符号的“段类型”。本例中，所有的外部符号都是CODE类型。

接下来是符号定义，第10行把CR定义为回车符，其ASCII码是0DH。第11行把符号EPROM定义为CODE类型的段。回顾前面提到的SEGMENT伪指令的作用，它仅仅用来定义符号的段类型，没有其他的功能。

第13行的RSEG指令表示可重定位的代码段EPROM的开始。随后的指令、数据常数定义等内容将被存放到EPROM代码段。

第14行的标号是MAIN，程序于此处开始，第1条指令调用子例程INIT，初始化8051的串行端口。该条指令汇编后的操作码位于列标OBJ对应的位置处，第1个字节是操作码（12H，对应LCALL），第2个字节和第3个字节本应该是子例程的地址，但此处都是0000H，且其后带有F标志，这表明子例程INIT的地址暂时还不能确定。

链接器/定位器在链接模块时，必须重定位该段。请注意，LOC列处的内容也是0000H。由于EPROM段是一个可重定位的段，因此，在汇编的时候，无法确定EPROM段的起始地址。在此情况下，生成列表文件的时候，所有的可重定位段的起始地址均以0000H作为起始地址。

15行~19行是程序余下的指令。把提示符字符串的起始地址置入DPTR，然后调用子例程OUTSTR，就会在VDT上显示提示符。由于OUTSTR、INLINE、OUTLINE等子例程没有定义在ECHO.SRC文件中，读者只能从子例程的名称以及注释行来猜测这些子例程的功能。

提示符是以NULL结尾的ASCII码字符串，在EPROM段的第21行用DB指令定义了该字符串。由于提示符的各个字节都是常数（固定不变的），所以尽管提示符是数据，也可以将其定义在代码空间内。提示符以回车符开头，目的是确保每次输出的时候，都处在新的一行上（在本例中，假设VDT会将CR转换为CR/LF）。

ECHO.LST列表文件下端的符号表，列出了ECHO.SRC源程序文件中的所有符号和标号。由于EPROM段是可重新定位的，而各个子例程都定义在其他模块中，因而符号表的VALUE列所包含的信息很少。但符号EPROM的值表示EPROM段的长度，在本例中为24H，所以EPROM段的长度是36B。

7.8.2 IO.LST

图7-8b展示了列表文件IO.LST的内容，主要包括相关的输入/输出子例程。在第6行定义该模块名称为SUBROUTINES。第7行和第8行声明了所有的子例程的名称都是PUBLIC属性的符号，表明这些子例程可用于其他模块。需要注意的是，尽管MAIN模块中只调用4个子例程，但所有的子例程都被声明为PUBLIC，这样做有利于程序扩展，因为当新的模块添加进来时有可能会调用这些子例程。

第13行和第16行定义了几个符号。EPROM在这里又一次被用作代码段的名称。其他段用在该模块之中。第17行定义了一个内部数据段ONCHIP。

从第20行起依次是各子例程的代码。本例中的注释很简洁，但通常会给出子例程功能属性的详细描述，提供这些信息对于将来调用这些子例程是很有帮助的。例如，注释块内可包含子例程输入和输出条件。

最后一个子例程后是利用ONCHIP段，在内部RAM中建立1个缓冲区。ONCHIP段采用RSEG伪指令（第82行）定义起始地址，缓冲区由DS（定义存储区）伪指令（第83行）建立，长度由符号LENGTH（第14行，值为40）确定。符号LENGTH在源文件中的定义位置和段ONCHIP在代码中的实例位置两者之间相距较远，但完全可以将这两个符号定义在其他位置，比如说在子例程时INLINE（该子例程用到了这两个符号）之前或之后。

从程序清单中可以看出，和EPROM段一样，LOC列下的第83行处，ONCHIP段

的初始地址也为0000H。同样的，ONCHIP段在存储器空间的实际地址要到链接的时候才能确定下来（见下文）。在IO.LST文件中，字母F出现在很多源代码行里。该标志的意思是，该行指令中包含了1个在汇编的时候不能确定值的符号。而当前目标文件中，汇编器暂且把这些符号的值以0来表示，但在链接/定位操作时，会用该符号的实际值来替代这些临时的0值。

7.8.3 EXAMPLE.M51

图7-8c展示了由链接器/定位器RL51所生成的列表文件EXAMPLE.M51的内容。该文件的前面是由ECHO.OBJ和IO.OBJ建立EXAMPLE.M51的RL命令行（注意，文件中增添了RL51程序的路径）：

```
RL51 ECHO.OBJ, IO.OBJ TO EXAMPLE CODE (EPROM (8000H)) & DATA
(ONCHIP (30H))
```

在RL命令之后列出了各个目标模块，彼此之间以逗号隔开，且按照待链接的顺序排列。输入列表之后是可选用的控制项TO EXAMPLE，该控制项给出了由RL51建立的、采用绝对地址的目标模块的名字。如果省略该控制项，链接器默认将输入列表中第1个文件的名字用做目标模块名（但略去了扩展名）。在本例中，根据控制项，列表文件命名为 EXAMPLE.M51。其后是和定位相关的控制项，CODE和DATA指明了相关类型的绝对地址段的名称和起始地址。本例中，代码段EPROM的起始地址为8000H，数据段ONCHIP的起始地址是30H（定义在8051的内部RAM）。

紧跟在命令调用行后面的是经由RL51处理的各个输入模块。在本例中，只包含2个文件（ECHO.OBJ与IO.OBJ）和2个模块（MAIN与SUBROUTINES）。注意：此处的文件名和模块名不同，如果在源文件中没有用NAME伪指令定义模块名，在默认情况下，模块和文件同名。

接着是链接映射表，给出了EPROM和ONCHIP两个段的类型、起始地址和长度（十六进制形式）。ONCHIP为数据段，起始地址是30H，长度为28H（40B），EPROM为代码段，起始地址为8000H，长度为67H（103B）。

EXAMPLE.M51文件的最后是符号表，表中按所在模块列出了所有在程序中用到的符号和标号，所有地址都是绝对地址。值得注意的是，只有在源文件的开头使用了$DEBUG汇编控制项后，RL51才会在.M51文件里建立相应的符号表。从符号表中可以查到，INIT子例程（在前面的ECHO.LST文件中，该符号的地址还是空缺的）位于SUBROUTINES模块的8024H地址。链接/定位器还负责把各个模块中CALL INIT指令的地址部分替换为INIT子例程的绝对地址，例如MAIN模块。知道了符号对应的绝对地址，在调试程序的时候特别有用。如果发现1个错误，可以直接在程序的二进制代码上临时打个“补丁”，修改某些字节的内容后，重新执行程序。如果错误被修正，再对源程序做相应的修改。

7.9 宏

作为本章最后一节，再次讨论ASM51汇编器。ASM51的宏处理实际上就是“字符串替代”功能。宏允许把频繁使用的某段代码定义成1个简单的助记符，之后可在程序中的任何位置插入该助记符，完成同样的功能。在编程过程中应用宏处理的方法将会大大地加强扩展8051处理问题的能力。在源程序的任何位置都可以定义宏，定义之后，可以像使用普通指令那样使用宏。宏定义的语法如下：

183

```
%*DEFINE (call_pattern) (macro_body)
```

一旦定义完成，call_pattern 就类似于指令的助记符，其使用方法和普通的汇编语言指令相似，将其放在程序的助记符字段位置即可，但和真正的指令有点不同，使用时需要在宏名称前面加上百分符号%（宏标志）。汇编器在编译源程序的时候，会逐字符地以宏的定义体代替call_pattern。宏并不神秘，只不过提供了一种方法，用简单易记的助记符代替一段繁琐的指令序列。在替代的时候，遵循逐字符替代原则，不会多加一个字符，也不会落下一个字符。

例如，在源程序的开头定义如下一个宏：

```
%*DEFINE  (PUSH_DPTR)
            (PUSH DPH
             PUSH DPL
            )
```

在后面用到了宏：

```
%PUSH_DPTR
```

在.LST列表文件中，该宏被替代为

```
PUSH DPH
PUSH DPL
```

上面例子是宏的一个典型应用。由于8051的栈相关指令仅支持直接寻址方式，而将数据指针压栈需要两条PUSH指令（分别对应DPTR的高位和低位字节）。同样也可以定义1个宏来实现DPTR的出栈功能。

总结起来，使用宏有以下几条明显的优势。

- 增强了源程序的可读性，因为宏助记符比起其所替代的汇编语言指令，在形式上更能表明其所完成的操作的功能和目的。
- 缩减了源程序规模，减少了文字录入的工作量。
- 减小了出错机率。
- 使编程者从低水平的细节处理过程中解放出来。

后两条彼此相关。一旦某个宏被定义且经过调试之后，便可以在程序中随意使用，无需担心出错。以上面的PUSH_DPTR宏为例，如果直接使用PUSH和POP指令，程序员可能会在不经意间颠倒出栈和进栈的顺序（先压栈的是高位字节还是低位字

节），从而造成错误。反之，如果使用了宏，在编写宏的时候就已经考虑到了压栈和出栈顺序的细节。因此当定义、调试完成之后，即可随意使用且无需担心出错。

由于宏的替换过程是逐字符进行的，因而在定义宏的时候，要注意回车符、制表符等符号的使用，以确保在替换后，宏语句和上下文的其余汇编语言程序能够很好地对齐。要想做到这一点，一般需要做些试验和修正。 184

ASM51的宏处理功能包含若干优点，如可用于实现参数传递、局部标号、重复操作、汇编流程控制等。下面将讨论这些内容。

7.9.1 参数传递

要想使宏能从主程序接受参数，需要采用如下的宏定义格式：

```
%*DEFINE (macro_name (parameter_list)) (macro_body)
```

如果定义了一个宏：

```
%*DEFINE (CMPA# (VALUE))
        (CJNE A,#%VALUE,$ + 3
        )
```

那么宏调用

```
%CMPA# (20H)
```

展开后，在.LST文件中对应的指令就是：

```
CJNE A,#20H,$ + 3
```

虽然8051不包含1条可以直接对累加器进行比较的指令，很容易用CJNE指令和$+3组合来实现此功能，其中$+3是紧接CJNE的下一条指令的地址，这里被用作条件跳转的目标地址。相比具体的指令，程序员更容易记住宏的助记符CMPA#。此外，用宏来代替具体的指令后，省掉了程序员对诸如$+3这样一些符号细节的关注。

再看一个例子。在很多编程场合下，8051如果包含实现下面功能的指令，那实在是编程者的幸运：

```
JUMP IF ACCUMULATOR GREATER THAN X
JUMP IF ACCUMULATOR GREATER THAN OR EQUAL TO X
JUMP IF ACCUMULATOR LESS THAN X
JUMP IF ACCUMULATOR LESS THAN OR EQUAL TO X
```

但8051不包含可以直接实现上述功能的指令。虽然程序设计者可以用CJNE指令加上JC或JNC等指令的组合完成上述功能，但其中的编程细节很麻烦。假设有个例子：程序就跳转目标地址为标号GREATER_THAN处；在累加器中的存放着1个大于“Z”（5AH）的ASCII码。下面的指令序列能实现这个任务：

```
CJNE    A,#5BH,$ + 3
JNC     GREATER_THAN
```

CJNE指令把累加器的值和5BH（即“Z”+1）相减，并根据相减结果设置进位标志C。 185

如果累加器的值在00H到5AH范围内（包括5AH），CPU将进位标志置1，即C=1（注意：5AH－5BH<0，所以C=1；但5BH－%BH=0，所以C=0）。如果累加器的值不小于5BH（例如为5BH、5CH、5DH等，直到FFH），CPU将进位标志清零，即C=0，程序跳转到标号GREATER_THAN处去执行。一旦弄清楚了指令的实现细节，可以将这段指令序列定义成1个宏，取个简单恰当的助记符名字，然后用宏去替代相应的指令序列。如下指令是1个实现“大于即跳转”功能的宏的定义过程：

```
%*DEFINE (JGT(VALUE, LABEL))
            (CJNE A,#%VALUE+1,$+3  ;JGT
            JNC    %LABEL
            )
```

如果需要测试累加器中存放的ASII码的值是否大于Z，如前所述，可以调用宏来实现：

```
%JGT ('Z',GREATER_THAN)
```

ASM51在编译源程序时，把上述宏调用展开如下：

```
CJNE   A,#5BH,$+3      ;JGT
JNC    GREATER_THAN
```

本例中的宏JGT很好地展示了宏的强大功能。使用宏给编程者带来的好处是，不但可以把常用的指令序列用1个含义清晰的助记符来代替，而且可以避免过多处理一些琐碎且容易造成潜在错误的细节。

7.9.2 局部标号

局部标号在宏的内部使用，其格式如下：

```
%*DEFINE(macro_name [(parameter_List)])
            [LOCAL list_of_local_labels] (macro_body)
```

请看下面的宏定义例子：

```
%*DEFINE (DEC_DPTR) LOCAL SKIP
          (DEC DPL              ;DECREMENT DATA POINTER
           MOV A,DPL
           CJNE A,#0FFH,%SKIP
           DEC DPH
%SKIP:     )
```

调用格式如下：

```
%DEC_DPTR'
```

186

ASM51把上面的宏调用展开为：

```
          DEC    DPL             ;DECREMENT DATA POINTER
          MOV    A,DPL
          CJNE   A, #0FFH, SKIP00
          DEC    DPH
SKIP00:
```

注意，一般为了防止局部标号和源程序中的同名标号发生冲突，所以在宏展开的时

候，ASM51会在局部标号后添加数字序号，以示区别。此外，如果下一次宏调用中用到了同名的局部标号，ASM51负责给该标号添加新的数字序号，以防相互混淆。

需要指出的是，上述定义的宏潜伏了一个“副作用”。宏DEC_DPTR中用累加器A来临时存放DPL，倘若在调用该宏的代码段中，A还有其他用途，但是在调用宏之后，A的值就被改变了，这很可能在程序执行时导致错误。要避免这个错误，可以把A的当前值保存到栈中。修改后的DEC_DPTR宏的定义如下：

```
%*DEFINE (DEC_DPTR) LOCAL SKIP
         (PUSHACC
         DEC DPL             ;DECREMENT DATA POINTER
         MOV A,DPL
         CJNE A,#0FFH,%SKIP
         DEC DPH
%SKIP:   POP ACC
         )
```

7.9.3 重复操作

下面是1个内嵌（预定义）的宏，其格式为：

```
%REPEAT (expression) (text)
```

例如，要在代码段中添加100个NOP指令，程序如下：

```
%REPEAT (100)
(NOP
)
```

7.9.4 控制流操作

ASM51提供控制流宏定义，用以实现代码段的条件汇编。格式如下：

```
%IF(expression) THEN (balanced_text)
[ELSE (balanced_text) ]FI
```

例如，

```
INTERNAL    EQU 1          ;1 = 8051 SERIAL I/O DRIVERS
                           ;0 = 8251 SERIAL I/O DRIVERS
            .
            .
            %IF (INTERNAL) THEN
(INCHAR:    .              ;8051 DRIVERS
            .
OUTCHR:     .
            )ELSE
(INCHAR:    .              ;8251 DRIVERS
            .
OUTCHR:     .
            .
            )
```


在本例中，符号INTERNAL的值若是为1，则选择8051串行端口的I/O子例程；若为0，则选择8251的外部UART的子例程。宏IF指示ASM51汇编器在二者中选其一，跳

过另一个。程序中其他位置如果用到INCHAR和OUTCHR等子例程，则无需再考虑硬件的配置问题。只要正确设置了INTERNAL的值，汇编器就会选择正确的子例程加以汇编。

小结

本章主要介绍了8051汇编语言的基本概念，包括标准的程序格式以及如何利用汇编伪指令和宏的方法来增强程序的可读性、缩减代码规模、规范程序等问题。

随着程序规模的增大，迫切需要有一种更适于结构化编程、更接近英语表述风格的其他有别于汇编语言的编程途径，C语言就是一个比较理想的选择，因此，下一章将讨论8051的C语言编程。

习题

7.1 重写如下各条指令，其中操作数以二进制代码形式来表示。

```
MOV    A,#255
MOV    A,#11Q
MOV    A,#1AH
MOV    A,#'A'
```

7.2 重写如下各条指令，其中操作数以十六进制代码形式来表示。

```
MOV    A,#255
MOV    A,#-3
MOV    A,#'Z'
MOV    A,#'33Q'
MOV    A,#'$'
MOV    A,#64
```

7.3 请找出如下指令的错在何处。


```
ORL    80H,#F0H
```

7.4 请指出下列符号的错误所在。

```
?byte.bit
@GOOD_bye
1ST_FLAG
MY_PROGRAM
```

7.5 重写如下各条指令，要求表达式的值以十六进制代码的形式来表示。

```
MOV   DPTR, #'0' EQ 48
MOV   DPTR, # HIGH'AB'
MOV   DPTR, #-1
MOV   DPTR, #NOT(257 MOD 256)
```

7.6 重写如下各条指令，要求各个源地址以十六进制形式来表示。

```
MOV    A,   PSW
```

```
MOV    A,   P0
MOV    A,   DPH
MOV    A,   TL0
MOV    A,   IP
MOV    A,   TMOD
```

7.7 重写如下各条指令，要求将各个十六进制的源地址以符号形式表示。

```
MOV    A,   0B0H
MOV    A,   99H
MOV    A,   82H
MOV    A,   8BH
MOV    A,   0A8H
MOV    A,   0D0H
```

7.8 ASM51汇编器中，什么是“段类型”？各自对应什么存储空间？

7.9 如何定义位于外部数据空间且可重定位的段？如何选择该段，如何在该段中创建1个100B的缓冲区？（假设段名为OFFCHIP，缓冲区名为XBUFFER。）

7.10 某程序需要5个状态位（FLAG1~FLAG5）。如何在一个采用绝对地址、且起始位地址为08H的位数据段中定义这5个位？它们所属的字节地址是多少？

7.11 请给出两条在汇编语言程序中应用EQU伪指令的好处。

7.12 DB和DW伪指令有什么区别？

7.13 下面各条伪指令对内存布局有什么影响？

```
ORG   0FH
DW    $ SHL 4
DB    65535
DW    '0'
```


7.14 选择1个采用绝对地址的代码段要采用什么伪指令？

7.15 1个名为ASCII的文件中包含33条EQU伪指令，部分伪指令如下：

```
NUL    EQU    00H              ; NULL BYTE
SOH    EQU    01H              ; START OF HEADER
:
US     EQU    7FH              ; UNIT SEPARATOR
DEL    EQU    7FH              ; DELETE
```

不把上述定义的符号插入到其他文件中，怎样才能使得这些符号为另一个文件（源程序）所知？

7.16 为了使得打印输出的列表文件显得整齐，最好让每个子例程位于列表文件页的开始处，如何才能达到这个目的？

7.17 定义1个宏，该宏可用来把外部数据空间的1段存储空间填以同一个常数。宏的参数有存储空间的起始地址和长度，以及用来填充的数据常数。

7.18 定义宏，完成如下功能：

JGE——如果累加器A的值大于或等于VALUE就跳转。

JLT——如果累加器A的值小于VALUE就跳转。

JLE——如果累加器A的值小于或等于VALUE就跳转。

JOR——如果累加器A的值处于LOWER和UPPER之外就跳转。

7.19 定义名为CJNE_DPTR的宏，完成如下功能：如果DPTR的值不包括VALUE，就跳转到标号LABEL处执行。要求在定义宏的时候，要确保不会改变任何寄存器和存储器空间的初始值。

190

第8章　8051的C语言编程

8.1　引言

在本书前面的章节里，一直探讨如何通过汇编语言同8051打交道的问题。事实上，还存在着另外一种和8051沟通的途径，即8051 C语言。当程序达到一定的复杂程度之后，8051 C语言往往是更理想的选择。本章把8051 C语言作为汇编语言编程的备选替代方案来介绍给读者。在程序开发时，究竟选择哪种语言最终由编程者自己决定。影响编程语言选择的一般因素有期望速度、代码规模和编程难易程度等。本章意在介绍8051 C语言编程的基本问题，并假设读者已经熟悉传统的C语言编程，因为现在C已经是非常普及且广泛使用的编程语言了。

8.2　8051中采用C语言的优缺点

对于8051，采用C编程同汇编语言比较主要优势有以下几点。

- 能提供C这样的高级结构化编程语言的所有优点，包括编写子例程以及在后面的9.2节中所讨论的更多的编程技术细节。
- 编程者无需对8051硬件结构以及编译操作的细节有特别全面的了解。
- 使得代码容易编写，尤其在针对较大规模的复杂程序时。
- 增加了源代码的可读性。

然而，8051 C和传统C语言很相似，所以在享受其优点的同时必然也要忍受其以下缺点。

- 具有高级结构化编程语言的共有缺点，详见9.2节。
- 通常情况下在编译后会产生大量的机器码。
- 削弱了编程者的直接硬件控制能力。

为了对8051 C和汇编语言有个直观的比较，下面把第4章的例4-5的例程改用8051 C语言编写。


```
sbit portbit = P1^0; /* Use variable portbit to refer to P1.0 */
main( )
{
TMOD = 1;
while (1)
        {
        TH0 = 0xFE;
        TL0 = 0xC;
        TR0 = 1;
```

```
        while (TF0 != 1);
        TR0 = 0;
        TF0 = 0;
        portbit = !(P1^0);
        }
    }
```

注意，该程序和例4-5的汇编语言程序的代码行数几乎相同。二者主要区别在程序的可读性方面。由于C版程序看起来更接近于人类语言，所以比汇编语言的可读性要好很多。这一点有助于编程者利用，C开发比较复杂的应用程序。汇编版的程序更接近于机器码，因此可读性不好，通常用汇编语言开发的程序编译后的机器码比较简洁。如本例所示，汇编版程序汇编后的机器码有83B，而C版程序编译后是149B，增加了79.5%。

图8-1是十分有趣的图片，描述了编程者采用C高级语言和汇编语言同8051沟通的过程，以及机器语言的举例说明。

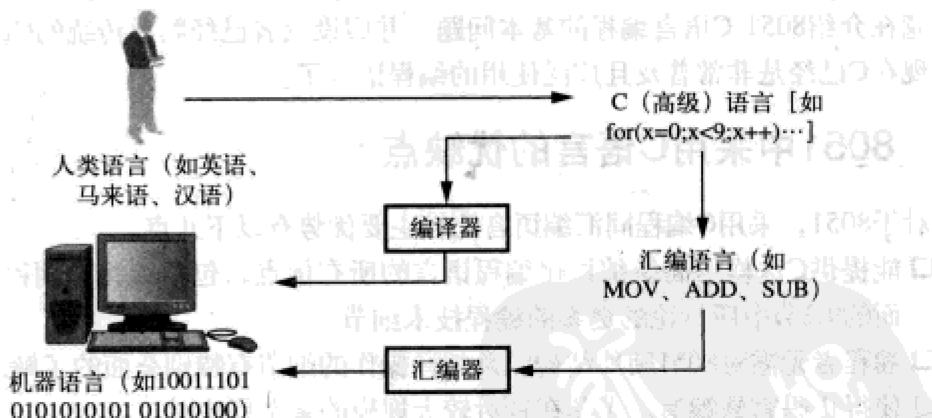


图8-1　人类语言、高级语言、汇编语言以及机器语言之间的转换

8.3　8051 C 编译器

由图8-1可知，将8051 C语言程序转换成机器语言需要1个编译器，就像汇编语言程序需要汇编器一样。编译器的基本功能和汇编器类似，只是前者更复杂，因为C和机器语言之间的差异要比汇编语言和机器语言之间的差异大得多。

192

目前有多种8051 C编译器，但它们所提供的基本功能大多相似。本书中的例程都已经在Keil软件公司①的8051程序集成开发环境Keil μVision 2 IDE（包括C51交叉编译器，详情请参阅附录H的μVision 2 IDE应用导引）上编译调试通过。交叉编译器可以在一些平台（诸如IBM兼容PC机）上运行，但由其编译生成的机器码可以在其他平台（如8051）上运行。

① Keil software, Inc., 1501 10th Street, Suite 110, Plano, TX 75074。网址：http://www.keil.com。

8.4 数据类型

8051 C除了一些扩展以及为了适于8051编程环境而做的某些改动之外，和传统的C语言是完全类似的。对8051 C编程者来说第一关心的就是数据类型。读者应该已经比较熟悉一些基本的C数据类型了，如int、char和float等，分别用于创建相应类型的变量用于存储整数、字符和浮点数。8051 C在支持这些基本的C数据类型的基础上，又增加了几个专用于8051的新数据类型。

表8-1列出了8051 C的常用数据类型，其中以黑体形式标出的是8051的专用数据类型。数据类型bit用于声明1个位于8051可位寻址空间（也就是内部RAM字节地址为20H~2FH或者位地址为00H~7FH）的位变量。显然，位变量只能存储0或1。例如，下面的C 语句将声明1个位变量flag并且初始化为0。

```
bit flag = 0;
```

sbit数据类型和bit数据类型有些相似，但其所声明的位变量位于特殊功能寄存器（SFR）。例如：

```
sbit P = 0XD0;
```

193

该C语句声明sbit类型的位变量P，并且将其位地址设置为D0H，这实际上是特殊功能寄存器PSW的最低有效位。需要注意的是这两种位数据类型（bit和sbit）的赋值运算符（=）之间的区别。在声明sbit数据类型变量的语句中，“=”表示sbit变量的地址；而在声明bit类型变量时，“=”表示bit变量的初始值。

表8-1 8051 C语言的数据类型

数据类型	位数	字节数	取值范围
bit	1		0~1
signed char	8	1	−128~+127
unsigned char	8	1	0~255
enum	16	2	−32 768~+32 767
signed short	16	2	−32 768~+32 767
unsigned short	16	2	0~65 535
signed int	16	2	−32 768~+32 767
unsigned int	16	2	0~65 535
signed long	32	4	−2 147 483 648~+2 147 483 647
unsigned long	32	4	0~4 294 967 295
float	32	4	±1.175 494E−38~±3.402 823E+38
sbit	1		0~1
sfr	8	1	0~255
sfr16	16	2	0~65 535

除了直接给sbit变量赋予位地址以外，还可以将预先定义的sfr变量的地址作为

基址，然后再标示出欲定义的sbit变量在sfr变量中的具体位置，通过这种方式来定义sbit变量的地址。例如：

```
sfr PSW=0XD0;
sbit P=PSW^0;
```

该声明首先定义PSW特殊功能寄存器变量，将其地址设为D0H，然后以此为基址，将sbit变量P设置在其最低有效位（第0位）。其中的“^”符号用于标明PSW的第0个位，这种表示方法具有通用性。

第3种方法，直接用常数字节地址作为基址，之后同样也将相应的相对位地址赋值给sbit变量。因此，第2种方法的2条语句可用如下语句替换：

```
sbit P=0XD0^0;
```

上面的C语句中已经同时用到了sfr数据类型，sfr常用于定义与SFR相关的字节（8位）变量。例如：


```
sfr IE=0XA8;
```

声明1个位于字节地址A8H的sfr变量IE，该地址（A8H）正是中断允许寄存器（IE）的地址。因此，sfr数据类型在某种意义上可以说是能够通过给SFR赋予名字的方式使得SFR更容易记忆。

sfr16数据类型和sfr很相似，但有一点区别，和sfr针对8位的特殊功能寄存器相比，sfr16针对的是16位的特殊功能寄存器。例如：

```
sfr16 DPTR=0X82;
```

声明1个16位的变量DPTR，其低位字节的地址是82H。回顾8051内部数据存储器的布局可知，82H是DPL特殊功能寄存器的地址。同样道理，sfr16数据类型使得通过其名称引用特殊功能寄存器要比通过地址引用要方便得多。通过声明sbit、sfr和sfr16变量的方式，使特殊功能寄存器的记忆和使用变得非常简单。否则，单纯通过地址方式访问特殊功能寄存器比较容易出错。

事实上，8051的所有特殊功能寄存器，包括特殊标志、状态以及位于可位寻址特殊功能寄存器的控制位，都已经被声明于1个头文件里，文件名为reg51.h，大多数8051 C编译器都打包有该文件。reg51.h文件的内容如图8-2所示。使用该头文件可以发现，通过中断允许寄存器的名称IE访问要比地址A8H简单得多，还有，使用DPTR访问数据指针寄存器也比地址82H简单。这些方式增强了8051 C程序的可读性和可管理性。

```
/*--------------------------------------------------------------------
REG51.H
Header file for generic 80C51 and 80C31 microcontroller.
Copyright (c) 1988-2001 Keil Elektronik GmbH and Keil Software,
Inc.
```

图8-2 reg51.h文件内容

```
All rights reserved.
--------------------------------------------------------------------------*/
/* BYTE Register */
sfr     P0        = 0x80 ;
sfr     P1        = 0x90 ;
sfr     P2        = 0xA0 ;
sfr     P3        = 0xB0 ;
sfr     PSW       = 0xD0 ;
sfr     ACC       = 0xE0 ;
sfr     B         = 0xF0 ;
sfr     SP        = 0x81 ;
sfr     DPL       = 0x82 ;
sfr     DPH       = 0x83 ;
sfr     PCON      = 0x87 ;
sfr     TCON      = 0x88 ;
sfr     TMOD      = 0x89 ;
sfr     TL0       = 0x8A ;
sfr     TL1       = 0x8B ;
sfr     TH0       = 0x8C ;
sfr     TH1       = 0x8D ;
sfr     IE        = 0xA8 ;
sfr     IP        = 0xB8 ;
sfr     SCON      = 0x98 ;
sfr     SBUF      = 0x99 ;
/* BIT Register */
/* PSW */
sbit CY           = 0xD7 ;
sbit AC           = 0xD6 ;
sbit F0           = 0xD5 ;
sbit RS1          = 0xD4 ;
sbit RS0          = 0xD3 ;
sbit OV           = 0xD2 ;
sbit P            = 0xD0 ;
/* TCON */
sbit TF1          = 0x8F ;
sbit TR1          = 0x8E ;
sbit TF0          = 0x8D ;
sbit TR0          = 0x8C ;
sbit IE1          = 0x8B ;
sbit IT1          = 0x8A ;
sbit IE0          = 0x89 ;
sbit IT0          = 0x88 ;
/* IE */
sbit EA           = 0xAF ;
sbit ES           = 0xAC ;
sbit ET1          = 0xAB ;
sbit EX1          = 0xAA ;
sbit ET0          = 0xA9 ;
sbit EX0          = 0xA8 ;
/* IP */
sbit PS           = 0xBC ;
sbit PT1          = 0xBB ;
```

图8-2 （续）

```
sbit PX1       = 0xBA ;
sbit PT0       = 0xB9 ;
sbit PX0       = 0xB8 ;
/* P3 */
sbit RD        = 0xB7 ;
sbit WR        = 0xB6 ;
sbit T1        = 0xB5 ;
sbit T0        = 0xB4 ;
sbit INT1      = 0xB3 ;
sbit INT0      = 0xB2 ;
sbit TXD       = 0xB1 ;
sbit RXD       = 0xB0 ;
/* SCON */
sbit SM0       = 0x9F ;
sbit SM1       = 0x9E ;
sbit SM2       = 0x9D ;
sbit REN       = 0x9C ;
sbit TB8       = 0x9B ;
sbit RB8       = 0x9A ;
sbit TI        = 0x99 ;
sbit RI        = 0x98 ;
```

195 ≀ 196

图8-2 （续）

8.5 存储类型及模式

8051有多种存储类型，包括内部程序和数据存储器以及外部程序和数据存储器等。在声明变量时，往往希望得知该变量存放于哪种类型的存储器。表8-2列出了8051常见的存储类型及相应细节的描述。

表8-2 8051 C语言的存储类型

存储类型	描述（范围）
code	程序（代码）存储区
data	直接寻址内部数据存储区（128B）
idata	间接寻址内部数据存储区（256B）
bdata	可位寻址内部数据存储区（16B）
xdata	外部数据存储区（64KB）
pdata	分页寻址外部数据存储区（256B）

在表8-2中给出的第1个存储类型是code。用于规定某变量存放于程序存储器，空间可达64KB。例如：

```
Char code errormsg[]= "An error occurred" ;
```

该语句声明1个名为errormsg的字符数组，存放在程序存储器中。

如果打算将1个变量存放到数据存储器，可以选择表8-2中的其余5种类型的数据存储器。虽然有很大的选择余地，但请注意：不同类型的数据存储器会影响访问

的速度和变量的位数。例如：

```
signed int data num1;
bit bdata numbit;
unsigned int xdata num2;
```

上面的第1行语句建立1个有符号整型变量num1，存放于内部数据存储区（即data存储类型，地址范围00H～7FH）。第2行语句声明1个位变量numbit，存放于可位寻址空间（字节地址为20H～2FH），存储类型为bdata。最后1行声明1个名为num2的无符号整型变量，位于外部数据存储区，存储类型为xdata。由于所有置于直接寻址内部数据存储区的变量的访问速度要比其他存储类型快得多，所以在程序对运行速度要求比较苛刻的情况下，尽量将变量定义为data存储类型。对于8052等系列的微控制器，其内部数据存储区达到了256B，可以用idata存储类型来声明。需要注意的是，由于idata类型采用的是间接寻址方式，所以其访问速度比data类型要慢。如果需要将变量存放于外部数据存储区，那么将其声明为pdata或xdata存储类型。其中pdata类型对应的是外部数据存储区的前256B（即第1个存储页），而xdata类型允许的寻址范围是整个64KB的外部数据存储空间。

如果在声明变量时忘了显式地规定其存储类型，或者编程者希望所有变量都配有默认存储类型（即不用逐个地显式声明），应该怎么办呢？在此情况下，可以通过存储模式做出相应的规定，表8-3列出了8051的存储模式。

表8-3 8051C语言的存储模式

存储模式	描 述
small	默认状况下将变量存放到可直接寻址的内部数据存储区(data)
compact	默认状况下将变量存放到外部数据存储区的前256B(pdata)
large	默认状况下将变量存放到外部数据存储区(xdata)

在程序中可以显式地通过C指令#program来设置选择某种存储模式。否则，默认的存储模式是small。该模式之所以是推荐模式，是因为，所有变量都被存放于内部数据存储区，拥有较快的访问速度。

compact存储模式将所有变量默认存放到外部数据存储区的第1页，而large模式在默认状态下将变量存放到整个64KB的外部数据存储空间。

8.6 数组

通常，在程序中将一些用于存储相同类型数据的变量构成一个有序集合，可以增加可读性。例如，表8-4中列出的十进制数字的ASCII码表。为了在8051 C程序中实现存储1个数据表的功能，就要用到**数组**。数组是相同数据类型变量的集合，其中的每个元素可以通过数组名和相应序号组合的方式来访问。

表8-4 十进制数字的ASCII表

十进制数字	十六进制表示的ASCII码值	十进制数字	十六进制表示的ASCII码值
0	30H	5	35H
1	31H	6	36H
2	32H	7	37H
3	33H	8	38H
4	34H	9	39H

198

存储十进制数字的ASCII码表的数组：

```
int table[10]=
{0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39};
```

需要注意的是，数组的元素之间用逗号隔开。在访问单个元素时，起始索引标号是0。也就是说，table[0]和table[9]分别对应第1个元素和最后1个（即第10个）元素。

8.7 结构

在某些情况下，需要将1组有相互关联的、但属于不同数据类型的变量组合在一起来描述对象。例如，某人的姓名、年龄和生日各属于不同的数据类型，但都是关于该人的一些描述信息。在此情况下就需要声明1个**结构**。结构是由不同数据类型的变量构成的数据组。例如：

```
struct person {
    char name[10];
    int age;
    long DOB;
    };
```

当声明1个结构之后，就可以利用其像数据类型那样建立结构变量了，成员包括姓名、年龄和生日。例如：

```
struct person grace={"Grace",22,01311980};
```

该语句建立名为grace的结构变量，用于存储Grace（人名）的姓名、年龄和生日。为了访问person结构变量的内部成员，可以采用变量名后跟随小数点以及成员名称的形式。所以，grace.name、grace.age、grace.DOB分别对应Grace的姓名、年龄和生日。

8.8 指针

当使用汇编语言进行8051编程时，常常用R0、R1和DPTR等寄存器存放某些数据的地址信息。如果要通过这些寄存器访问相应的数据，应该使用间接寻址方式。在这种情况下，利用R0、R1和DPTR来指向对应的数据，所以，其所起的作用相当

于指针。

在C语言中，可以通过特殊定义的指针变量实现数据的间接访问。一些学生在学习C语言的过程中逃避指针，认为其不容易理解。实际上，指针只是1种比较特殊的变量类型，普通类型的变量可以直接存储数据，而指针变量存储的是数据的地址。需要注意的是，无论使用的是一般变量还是指针变量，最终在程序中要访问的对象都是数据。在使用一般变量的情况下，是直接到相应的存储单元去取数据；在使用指针变量的情况下，在取数据之前需要先知道该数据的存储地址，然后以地址作为中间纽带再去访问数据。图8-3描述了利用一般变量和指针变量访问数据的操作过程。

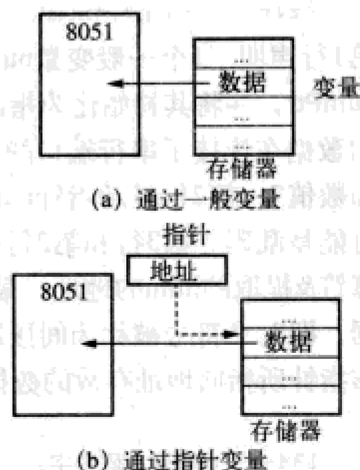


图8-3 访问数据的两种方式

声明指针变量的格式如下：

```
data_type  *pointer_name;
```

格式中的各部分分别是：

data_type	所定义的指针指向的数据类型
*	指针变量的标志
pointer_name	指针的名字

例如：

```
int  * numPtr;
int num;
numPtr=&num;
```

第1条指令声明1个名为numPtr的指针变量，指向整形数据。作为对比，第2行声明了1个名为num的一般整形变量。第3行将num的地址赋值给指针变量numPtr。如同本例一样，地址运算符&可用于获取任意变量的地址。需要注意的是，第3行的地址赋值完成后，numPtr指针的内容是num变量的地址，而不是该变量的数值。

上面的3行指令也可重写如下，也就是允许在声明指针变量的同时直接进行地址赋值：

```
int  num;
int *numPtr=&num;
```

为了进一步描述一般变量和指针变量二者之间的区别，请读者阅读下面这段程序，虽然其不是一个完整的C程序，但足可以用来说明问题：

```
int  num=7;
int *numPtr=&num;
printf("%d\n",num);
```

```
printf("%d\n",numPtr);
printf("%d\n",&num);
printf("%d\n",*numPtr);
```

第1行声明了1个一般变量num，并将其内容初始化为7。接着，声明了1个指针变量numPtr，并将其初始化为指向num变量的地址。再下面的4行使用printf()函数将相应的数据在连接于串行端口的显示终端上打印输出。其中第1行显示变量num的内容，即数值7。第2行显示指针numPtr，内容是num变量的地址，但在显示终端上看到的可能是乱码。第3行和第2行相同，也显示的是num变量的地址，此处是用取地址运算符&提取的num的地址。最后1行显示的是存储在numPtr指针所指向地址的实际数据，即7。*符号被称为间接运算符，当把它置于指针变量之前时，可以间接获得在该指针所指向地址存放的数据。所以，最终在显示终端上输出的内容为：

```
7
13452（或者是乱码数字）
13452（或者是乱码数字）
7
```

在稍后的8.10节中，将讨论完整的C程序，到时候读者可以将上面介绍的内容自行用程序的运行结果来验证。

8.8.1 指针的存储类型

考虑到指针也是一种变量，所以，它同样也有存储问题。在声明指针时，可以显式地对其存储类型做出相应的规定，例如：

```
int  * xdata numPtr=&num;
```

和前面的例子类似，此处也声明了指针变量numPtr，并且指向整型变量num。不同之处在于，在本例中显式地规定了该指针的存储类型，即在*号之后添加了xdata，表明numPtr存放在外部数据存储区（xdata），也可以说，该指针的存储类型是xdata。

8.8.2 定义数据存储类型的指针

可以更加详细地声明1个指针变量，例如：

```
int data * xdata numPtr = &num;
```

该语句也声明1个存储于外部数据存储区（xdata）的指针变量numPtr，而且其所对应的整型变量num存储在内部数据存储区（data）。*号前面的存储类型说明符data用于规定指针所指向的数据存储类型，而*号后面的存储类型说明符xdata用于规定指针变量的存储类型。

如果在1个指针变量的声明过程中，显式地规定了其所指向数据的存储类型，那么将其称为“定义数据存储类型的指针”。当做完定义数据存储类型指针的声明

后，其所指向的数据将按声明中所规定的存储类型存放。该类型的指针变量所占据的字节数同所规定的数据存储类型相关，可能是1或2字节。

8.8.3 未定义数据存储类型的指针

如果在声明1个指针变量时没有对相应的数据存储类型做出显式的规定，那么得到1个“未定义数据类型的指针变量”，这有点类似于通常的指针变量，即其所指向的数据可以存放在任何1种类型的存储区内。使用未定义数据类型指针变量的优点是，可以用于指向任意数据，因为没有对数据存储类型做出相应的规定和限制。未定义数据类型指针一般占据3个字节，比定义数据类型指针要多。而且一般情况下访问速度较慢，因为数据的存储类型一直到编译文件运行时才能最终确定。未定义数据类型指针的第1个字节表示其所对应数据的存储类型，见表8-5，5个数字分别对应不同的类型。第2和第3字节分别对应该指针所指向数据的高位字节地址和低位字节地址。

表8-5 未定义数据存储类型指针变量第1字节中的数值和数据存储类型的对应关系

数 值	数据存储类型
1	idata
2	xdata
3	pdata
4	data/bdata
5	code

未定义数据类型指针变量的声明过程和标准C很类似，例如：

```
int * xdata numPtr = &num;
```

注意，在该语句中没有对指针所指向的数据存储类型给出显式的规定。

8.9 函数

在应用汇编语言进行8051编程的过程中，读者已经体会过使用子例程（将具有代表性的、经常使用的系列指令集合在一起）的益处。同样，在8051 C语言中也有子例程的概念，但是已经不称为子例程了，而是称为函数。像传统C一样，这里的函数也需要声明和定义。函数的定义包括输入参数的类型和名称、输出参数（返回值）的类型，另外还有函数内容和功能的说明。

下面是函数定义的典型格式：

```
return_type function_name (arguments) [memory] [reentrant]
                                      [interrupt] [using]
{
        ...
}
```

这里的①

return_type	返回值（输出）的数据类型
function_name	由编程者规定的函数名
arguments	输入参数列表（包括类型和名称）
memory	存储模式（包括small，compact，large）
reentrant	是否为递归函数
interrupt	中断服务函数(ISR)的标识
using	选择工作寄存器组

举一个典型例子——求两数之和的函数。

```
int    sum (int a,int b)
{
         return a+b;
}
```

该函数名为sum，包括2个输入参数，且数据类型均为整型变量。函数的返回值也是整型的。用大括号将整个函数体括起来，函数的返回值很简单，是2个输入参数的代数和。在本例中忽略了4个选择项（包括memory，reentrant，interrupt和using）的声明，这意味着传递给该函数的参数将使用默认的small存储模式，而且被存放到内部数据存储区，还表明该函数不是递归函数，也不是中断服务函数。另外，该函数选择的工作寄存器组为第0组。

8.9.1 参数传递

在8051 C中，参数主要用于向函数传递数据，作为函数的输入变量使用。虽然函数参数存储的技术细节对编程者来讲是透明的，但一般不需要对这些专业知识有过多的关注。在本节中，为了使编译器的处理过程对我们更有益，将对参数的传递过程做简要讨论。在8051 C语言中，参数是通过寄存器和存储器进行传递的。通过寄存器进行参数传递的速度较快，而且这是默认的传递方式。关于利用寄存器传递参数的细节请读者参阅表8-6。

203

表8-6 利用寄存器传递参数

参数个数	Char/1字节指针	Int/2字节指针	Long/Float	普通指针
1	R7	R6 & R7	R4 ~ R7	R1 ~ R3
2	R5	R4 & R5	R4 ~ R7	
3	R3	R2 & R3		

由于8051只有8个寄存器，所以存在参数传递过程中寄存器不够用的问题。在这种情况下，剩余的参数需要通过固定的存储单元来传递。如果使用了NOREGPARM

① 其中的选择项需要用方括号分隔开。

控制指令，那么所有的参数都通过存储器传递，反之，如果使用REGPARM控制指令，那么所有的参数都通过寄存器进行传递。

8.9.2 返回值

和输入参数不同（既可以通过寄存器传递又可以通过存储器传递），函数的输出值必须通过寄存器传递。表8-7列出了不同类型的返回值所用到的寄存器。

表8-7 利用寄存器传递函数的返回值

返回类型	寄存器	描述
bit	进位标志（C）	
char/unsigned char/1-byte pointer	R7	
int/unsigned int/2-byte pointer	R6 & R7	最高有效字节位于R6，最低有效字节位于R7
long/unsigned long	R4 ~ R7	最高有效字节位于R4，最低有效字节位于R7
float	R4 ~ R7	32位IEEE格式
普通指针	R1 ~ R3	存储类型在R3中，最高有效字节位于R2，最低有效字节位于R1

8.10 8051 C语言实例

在本章的前半部分，简要讨论了利用8051 C语言编程的一些基本概念。在这一小节中，开始讨论如何编写各种控制程序。本节中讨论的所有8051 C程序都已经在Keil's μ Vision2（1种8051 集成开发环境（IDE），包括源代码编辑器、编译器和调试器）上面编译调试通过①。

8.10.1 第1个8051 C程序

如同一些结构化高级程序语言的入门教材一样，关于8051 C的第1个例程也是显示Hello World的简单程序。

204

```
#include <REG51.H>              /* SFR declarations */
#include <stdio.h>              /* Declarations for I/O
                                   functions(eg. printf) */

main ()
{
SCON = 0x52;                    /* serial port, mode 1 */
TMOD = 0x20;                    /* timer 1, mode 2 */
TH1 = -13;                      /* reload count for 2400
                                   baud */
TR1 = 1;                        /* start timer 1 */

while (1)                       /* repeat forever */
      {
```

① 该软件的免费演示版可以在KEIL公司的网址http://www.keil.com下载。

```
        printf ("Hello World\n");   /* Display "Hello World" */
        }
    }
```

该程序将在与8051串行端口连接的设备上连续显示Hello World信息，这里在默认状态下是模拟的串行窗口。关于μVision2 IDE的细节问题，请参阅附录H中Keil公司的帮助文档。

8.10.2 定时器

第4章讨论了如何使用8051的内置定时器，作为8051的汇编语言编程和C语言编程的对比，在这里将给出第4章中某些例子的C语言解决方案。

例8-1 方波发生器

编写程序在P1.0口建立尽可能高频的方波信号。

答案：

```
#include <REG51.H>          /* SFR declarations */
sbit portbit = P1^0;        /* Use variable portbit to refer to
                               P1.0 */
main ()
{
while (1)                   /* repeat forever */
        {
        portbit = 1;        /* set P1.0 */
        portbit = 0;        /* clear P1.0 */
        }
}
```

讨论： 该程序实际是例4-2汇编程序的C语言版本。程序中，端口P1.0需要重复地置位和清零，可以通过简单地向P1.0写1和写0来实现。由于P1已经声明在头文件REG51.H中，所以可以在主程序中将P1的最低有效位声明为1个sbit变量。

在利用指令产生精确的时间延迟和波形的应用场合中，有些问题需要注意。如果使用的是汇编语言，因为已经知道1条确定的指令的执行需要花费多少机器周期，所以可以容易地精确估算指令的执行时间。但如果使用C语言，就不能直接确定出其执行某条语句、某个循环或某个函数的执行时间了。C程序的执行时间和所采用的编译器有关系，因为不同的编译器对代码的优化水平不同，这就直接导致了生成的汇编代程序以及机器码也不相同。

205

解决这个问题的有效方法是，首先查找到某条确定的C语句所对应的汇编代码，然后再计算出执行这些汇编代码所需的机器周期。在μVision2 IDE中，可以对C程序执行反汇编操作，然后在反汇编窗口中观察汇编程序。与此相关的更详细信息，请参阅附录H或者Keil软件的帮助文档。

例8-2 10kHz方波

编写程序，利用定时器0在P1.0处产生10kHz的方波。

答案：

```
#include <REG51.H>              /* SFR declarations */
sbit portbit = P1^0;            /* Use variable portbit to
                                   refer to P1.0 */

main ()
{
TMOD = 2;                       /* 8-bit auto-reload mode */
TH0 = -50;                      /* -50 reload value in TH0 */
TR0 = 1;                        /* start timer 0 */

while (1)                       /* repeat forever */
        {
        while (TF0 != 1);       /* wait for overflow */
        TF0 = 0;                /* clear timer overflow flag */
        portbit = !(portbit);   /* toggle P1.0 */
        }
}
```

讨论：该程序实际上是例4-4的C版本。采用while语句建立了1个无限循环体，使用常数1作为while语句的条件，在C语言中，1用于表示逻辑真，0用于表示逻辑假。因此，在这种情况下，while的条件总为真，于是建立起了无限重复的循环体。

例8-3 1kHz方波

应用定时器0在P1.0产生1kHz的方波信号。

答案：

```
#include <REG51.H>              /* SFR declarations */
sbit portbit = P1^0;            /* Use variable portbit to
                                   refer to P1.0 */
main ()
{
TMOD = 1;                       /* 16-bit timer mode */
while (1)                       /* repeat forever */
        {
        TH0 = 0xFE;             /* -500 (high byte) */
        TL0 = 0x0C;             /* -500 (low byte) */
        TR0 = 1;                /* start timer 0 */
        while (TF0 != 1);       /* wait for overflow */
        TR0 = 0;                /* stop timer 0 */
        TF0 = 0;                /* clear timer overflow flag */
        portbit = !(portbit);   /* toggle P1.0 */
        }
}
```

206

讨论：该程序是例4-5汇编程序的C版本，而且和例8-2非常相似。

例8-4 蜂鸣器接口

已知，蜂鸣器连接在P1.7上，具有消除抖动功能的开关连接在P1.6上。要求编

程实现，当在P1.7探测到从1到0的电平跳变时，使蜂鸣器鸣响1s。

答案：

```
#include <REG51.H>              /* SFR declarations */

int hundred = 100;

sbit inbit = P1 ^ 6;            /* Use variable inbit to refer to
                                   P1.6 */
sbit outbit = P1 ^ 7;           /* Use variable outbit to refer to
                                   P1.7 */
unsigned char R7;               /* use 8-bit variable to represent
                                   R7 */

void delay(void);               /* Function prototype */
main ()
{
TMOD = 1;                       /* use timer 0 in mode 1 */
while (1)                       /* repeat forever */
      {
      while (inbit!=1);       /* wait for 1 input*/
      while (inbit==1);       /* wait for 0 input*/
      outbit = 1;             /* turn buzzer on */
      delay();                /* wait for 1 second */
      outbit = 0;             /* turn buzzer off */
      }
}
void delay(void)
{
R7 = hundred;
do
       {
       TH0 = 0xD8;              /* -10000 (high byte) */
       TL0 = 0xF0;              /* -10000 (low byte) */
       TR0 = 1;
       while (TF0 != 1);
       TF0 = 0;
       TR0 = 0;
       R7 -= 1;
       }
while (R7!=0);
}
```

207

讨论：该例程由delay()和主函数构成。由于delay函数在主函数之后定义，所以在主函数之前需要增加1条函数原型语句，通知编译器存在1个delay函数。R7在头文件REG51.H中未定义，所以需要在使用（R7用在函数delay中）之前定义。之所以这样做，完全是为了和例4-7的汇编程序在形式上保持一致。在实际的C程序中，一般不会使用通用寄存器的符号R0~R7，毕竟定义1个8位的变量作为替代也是很容易的事情。因为R0~R7在编译器中要用于传递参数，所以最好在C程序中不使用这些符号，以免造成混淆。

8.10.3　串行端口

本节将给出一些在第5章中出现的汇编语言例程的C语言版本。

208

例8-5 初始化串行端口

编写程序，将串行端口初始化为8位UART工作模式，波特率为2400，要求使用定时器1提供波特率时钟。

答案：

```
#include <REG51.H> /* SFR declarations */

main ()
{
SCON = 0x52;          /* serial port, mode 1 */
TMOD = 0x20;          /* timer 1, mode 2 */
TH1 = -13;            /* reload count for 2400 baud */
TR1 = 1;              /* start timer 1 */
}
```

例8-6 输出字符子例程

编写OUTCHR函数，通过8051的串行端口发送累加器中的7位ASCII码，且将奇数奇偶校验位作为第8位数据一并发送，要求在调用完该函数之后，累加器恢复到调用之前的状态。

答案：

```
#include <REG51.H>        /* SFR declarations */

sbit AccMSB = ACC ^ 7;  /* Use variable AccMSB to refer to
                           ACC.7 */

void OUTCHR(void)
{
CY = P;                 /* put parity bit in C flag */
CY = !CY;               /* change to odd parity */
AccMSB = CY;            /* add to character code */
while (TI != 1);        /* Tx empty? no: check again */
TI = 0;                 /* yes: clear flag and */
SBUF = ACC;             /* send character */
AccMSB = 0;             /* strip off parity bit */
}
```

讨论：在本程序中需要访问累加器ACC的最高有效位，所以声明了1个sbit变量AccMSB，代表累加器的最高有效位ACC.7。

例8-7 输入字符子例程

编写程序INCHAR，实现通过8051的串行端口输入7位的ASCII码，并存放到累加器中。要求接受字符的第8位是ASCII码的奇数奇偶校验位，如果出现了奇偶校验差错，将进位标志置1。

209

答案：

```
#include <REG51.H>        /* SFR declarations */

sbit AccMSB = ACC ^ 7;  /* Use variable AccMSB to refer to
                           ACC.7 */

void INCHAR(void)
```

```
{
while (RI != 1);          /* wait for character */
RI = 0;                   /* clear flag */
ACC = SBUF;               /* read char into accumulator */
CY = P;                   /* for odd parity in accumulator, P
                             should be set */
CY = !CY;                 /* complementing correctly indicates if
                             "error" */
AccMSB = 0;               /* strip off parity */
}
```

8.10.4 中断

通过8.9节对函数的讨论可知，中断服务程序（ISR）的编写和普通函数十分相似，但有一点区别，在中断服务程序的定义过程中需要用“中断”声明。下面将第6章中与中断相关的部分程序改用8051 C来编写，通过这种方式学习中断的8051 C编程。

例8-8 利用定时器中断产生方波

编写程序实现，利用定时器0和中断在P1.0产生10kHz的方波。

答案：

```
#include <REG51.H>        /* SFR declarations */

sbit portbit = P1 ^ 0;    /* Use variable portbit to refer
                             to P1.0 */

main ()
{
TMOD = 0x02;              /* timer 0, mode 2 */
TH0 = -50;                /* 50 µs delay */
TR0 = 1;                  /* start timer */
IE = 0x82;                /* enable timer 0 interrupt */
while(1);                 /* repeat forever */
}

void T0ISR(void) interrupt 1
{
portbit = !portbit;       /* toggle port bit P1.0 */
}
```

讨论：该程序介绍了中断函数的使用方法，这是一种特殊的函数类型，只要相应的中断发生，中断函数立即会自动运行。需要注意的是，在中断函数的声明语句中，含有1个中断源编号，本程序里是1，对应定时器/计数器0中断源。表8-8给出了8051各种中断源的编号、类型和向量地址。

例8-9 利用定时器中断产生2种方波

编程实现，利用中断同时在P1.7和P1.6产生7kHz和500kHz的方波。

答案：

```
#include <REG51.H>      /* SFR declarations */

sbit portsev = P1 ^ 7; /* Use variable portsev to refer to
                           P1.7 */
sbit portsix = P1 ^ 6; /* Use variable portsix to refer to
                           P1.6 */
main ()
{
TMOD = 0x12;            /* timer 1, mode 1; timer 0, mode 2 */
TH0 = -71;              /* 7kHz using timer 0 */
TR0 = 1;                /* start timer */
TF1 = 1;                /* force timer 1 interrupt */
IE = 0x8A;              /* enable both timer interrupts */
while(1);               /* repeat forever */
}

void T0ISR(void) interrupt 1
{
portsev = !portsev;     /* toggle port bit P1.7 */
}

void T1ISR(void) interrupt 3
{
TR1 = 0;
TH1 = 0xFC;             /* 1ms high time & */
TL1 = 0x18;             /*        low time */
TR1 = 1;
portsix = !portsix;     /* toggle port bit P1.6 */
}
```

讨论：在该例程中，2个中断函数分别用于响应定时器0和定时器1中断。查阅表8-8可知，这两个中断源对应的中断编号分别为1和3。

表8-8 标准8051的中断源和中断编号

中断编号	描　述	向量地址
0	外部中断0	0003H
1	定时器/计数器0中断	000BH
2	外部中断1	0013H
3	定时器/计数器1中断	001BH
4	串行中断	0023H

例8-10 利用中断控制字符输出

编写程序，利用中断方法向连接于8051串行端口的终端连续发送ASCII码表（控制符除外）。

答案：

```
#include <REG51.H> /* SFR declarations */

main ()
{
TMOD = 0x20;          /* timer 1, mode 2 */
TH1 = -26;            /* 12000 baud reload value */
TR1 = 1;              /* start timer */
SCON = 0x42;          /* mode 1, set TI to force 1st interrupt */
```

```
ACC = 0x20;          /* send ASCII space first */
IE = 0x90;           /* enable serial port interrupt */
while(1);            /* repeat forever */
}

void SPISR(void) interrupt 4
{
if (ACC == 0x7F)     /* if finished ASCII set */
     ACC = 0x20;     /*     reset to space */
SBUF = ACC;          /* send char. to serial port */
ACC = ACC + 1;       /* increment ASCII code */
TI = 0;              /* clear interrupt flag */
}
```

讨论：该程序利用1个中断函数去响应串行口的中断请求，中断编号为4。需要注意的是，虽然对于中断函数的命名没有什么限制，但是通常的作法是取比较直接明了（同相应的中断源相关）的名称，例如SPISR和T0ISR等。

例8-11　锅炉控制器

设计基于8051的锅炉控制器，要求利用中断方法将1个大楼的温度控制在20°C±1°C。

答案：

```
#include <REG51.H>      /* SFR declarations */
sbit outbit = P1 ^ 7; /* Use variable outbit to refer to P1.7
                         */
sbit hotbit = P3 ^ 2; /* Use variable hotbit to refer to P3.2
                         */

main ()
{
IE = 0x85;              /* enable external interrupts */
IT0 = 1;                /* negative edge triggered */
IT1 = 1;
outbit = 1;             /* turn furnace on */
if (hotbit != 1)        /* if T > 21 degrees, */
     outbit = 0;        /*     turn it off */
while(1);               /* repeat forever */
}

void EX0ISR(void) interrupt 0
{
outbit = 0;             /* turn furnace off */
}
void EX1ISR(void) interrupt 2
{
outbit = 1;             /* turn furnace on */
}
```


讨论：通过该程序可以看到，编写响应外部中断请求的中断函数，同编写响应定时器中断及串行中断的函数是类似的，不同之处仅在于中断编号（可以在表8-8中查到）的差异。该例程中的中断函数的命名也是和相应的中断源紧密相关的。

例8-12 入侵报警系统

设计1个入侵报警系统，当接在 $\overline{\text{INT0}}$ 的门禁传感器探测到从高到低的电压跃变时，通过接到P1.7的扬声器产生400Hz、持续1s的报警声，要求采用中断方式实现此功能。

答案：

```
#include <REG51.H>      /* SFR declarations */

sbit outbit = P1 ^ 7; /* use variable outbit to refer to P1.7
                        */
unsigned char R7;      /* use 8-bit variable to represent R7 */

main ()
{
IT0 = 1;               /* negative edge activated */
TMOD = 0x11;           /* 16-bit timer mode */
IE = 0x81;             /* enable EXT 0 only */
while(1);              /* repeat forever */
}

void EX0ISR(void) interrupt 0
{
R7 = 20;               /* 20 x 5000 µs = 1 second */
TF0 = 1;               /* force timer 0 interrupt */
TF1 = 1;               /* force timer 1 interrupt */
ET0 = 1;               /* begin tone for 1 second */
ET1 = 1;               /* enable timer interrupts */
                       /* timer interrupts will do the work */
}

void T0ISR(void) interrupt 1
{
TR0 = 0;               /* stop timer */
R7 = R7 - 1;           /* decrement R7 */
if (R7 == 0)           /* if 20th time, */
      {
      ET0 = 0;         /* disable itself */
      ET1 = 0;
      }
else
    {
    TH0 = 0x3C;        /* 0.05 sec. delay */
    TL0 = 0xB0;
    TR0 = 1;
    }
}

void T1ISR(void) interrupt 3
{
TR1 = 0;
TH1 = 0xFB;            /* count for 400Hz */
TL1 = 0x1E;
outbit = !outbit;      /* music maestro! */
TR1 = 1;
}
```

讨论：该例程实质是上文中几种中断函数的综合应用。

小结

本章介绍了利用C语言进行8051编程的一些概念，另外和前面章节相比，讨论了一些关于8051 C语言编程的实例（在前面的章节中是用汇编语言编程的）。无论是汇编还是C语言，二者都能完成编程任务，尤其是在需要复杂的组织化、结构化编程的场合。这是本书下一章将要讨论的主题。

习题

8.1 在数据类型方面，8051 C和传统的C语言有哪些差别?

8.2 假设需要声明1个变量用于在位地址20H存放1个位的数据，请用8051 C完成这个任务。

8.3 在可位寻址存储单元中，符号^有什么用途?

8.4 声明1个sbit变量有几种方法？各举1例进行解释说明。

8.5 SFR变量和无符号字符变量的区别是什么?

8.6 术语“存储类型”和“存储模式”有什么区别?

8.7 解释函数和中断函数的主要区别。

8.8 通过实例解释对“指针”概念的理解。

8.9 假设需要将某程序的大部分变量存放在外部数据存储器中，而将几个对访问速度要求严格的变量存放在内部数据存储器。简要论述如何达到此目的。 [214]

8.10 为什么在8051 C编程时使用累加器ACC是不可取的?

8.11 编写C程序实现，将从1~10这10个数字存放到程序存储器，之后，当P1.0为高（低）电平时，将其中的奇数（偶数）复制到外部数据存储器。

8.12 编写8051 C程序实现，将存放在内部数据存储器30H单元的ASCII字符串连续传送到外部数据存储器的1234H单元?

8.13 a. 在程序存储器中建立1个待查表，存储函数 $f(x)=e^{x+2}$的前10个值，其中$x=\{1, 2, \cdots, 10\}$。

b. 接下来，声明1个存放于间接访问的内部数据存储区的指针变量，通过该指针获得函数 $f(2)$的值，并将其在显示屏上打印输出。 [215]

第9章 程序结构和设计

9.1 引言

是什么因素决定了某程序优于另一个程序呢？除了诸如“运行”这样的简单条件之外，还有考虑其他诸多复杂因素，如维护要求、开发所用的计算机语言、技术文档的质量、开发周期、程序的大小、执行时间、可靠性、安全性等。本章介绍“好程序”的特点以及一些用于开发“好程序”的技术。首先介绍结构化程序设计技术。

结构化程序设计是一种对程序进行组织和编码的技术，它可以降低程序结构的复杂度，提高代码的清晰性，使得程序便于调试和修改。几乎所有的程序设计领域都强调结构化程序设计的重要性，本书也是如此。之所以如此重视结构化编程设计理念，是因为所有的程序都可以采用3种结构写出来。这听起来好像有些不太现实，但实际上，“顺序结构”、“循环结构”以及“选择结构”组合起来，就可以形成所有复杂的程序结构。因而仅用这3种结构即可实现所有的程序。程序控制流在3种结构里传递，而无需用无条件跳转指令（如Goto）转移到其他结构。每一种结构只有1个入口和1个出口。典型的结构化程序中包含层次化的子例程，所要求各个子例程只能有1个入口和1个出口。

本章主要介绍如何在汇编语言和8051 C语言的程序设计中采用结构化程序设计思想。高级语言（如Pascal和C）通过WHILE、FOR等形式的语句以及约定俗成的字符排列规则（也就是字符）来实现结构化程序设计，但汇编语言本身不存在类似的语言特性来帮助实现结构化程序设计。然而，如果程序员能在进行汇编语言程序设计的时候，秉承结构化程序设计的理念，最终收益是非常大的。但是，由于8051 C是C语言的扩展，继承了传统C语言的所有结构化编程技巧。所以，在本章中讨论汇编语言的结构化编程技术时，作为对照，将同时给出相应的8051 C程序，并分析其结构化特性。

本章的目的是指导程序设计者开发出好的汇编语言程序。在相关的例子里，都采用4种方式来解决问题：

- ❑ 流程图
- ❑ 伪码
- ❑ 汇编语言

❑ 8051 C语言

采用汇编语言来编程解决问题当然是最终目的，但在程序设计的开始阶段，流程图和伪码是很有用的工具。两者都是"可视化"的工具，便于程序员理解问题和构造程序。利用这两个工具可以把原来的问题由"怎么做"转换为"该做什么"，增加了可操作性。用流程图和伪码给出的解决方案，是和特定的机器不相关的通用"语言"，也就是说，在做流程图和用伪码编写程序时，无需考虑目标机器指令集的各种细节。

程序员通常没有必要同时使用伪码和流程图，该选择哪一种，主要由个人风格而定。图9-1中给出了最常用的流程图符号。

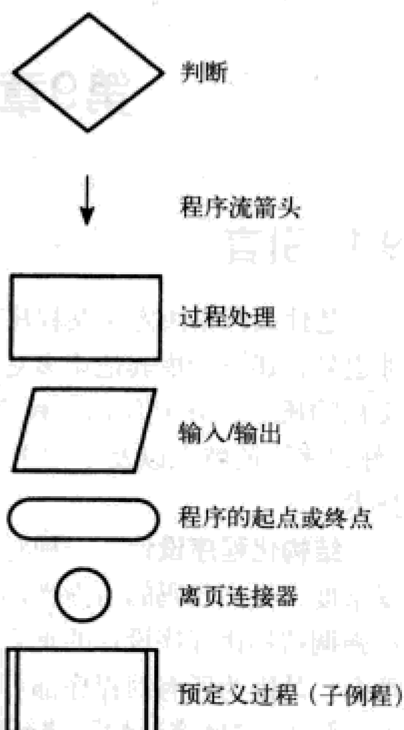


图9-1 流程图符号

就像"伪码"名字所透露的那样，伪码在一定程度上可称为一种计算机语言。在过去，伪码常作为一种方便的手段，用来非正规地描述程序设计问题的解决方案。本章中所采用的伪码，和Pascal以及C语言的语法相似，但同时加入了自然语言来描述程序中的操作。因而，在伪码中可能出现如下语句：

218

```
[get a character from the keyboard]
```

使用伪码的好处在于，把高级语言结构上的严谨和自然语言使用上的方便结合了起来。因此，在伪码中存在如下语句：

```
IF [condition is true]
      THEN [do statement 1]
      ELSE BEGIN
            [do statement 2]
            [do statement 3]
      END
```

或者

```
IF [the temperature is less than 20 degrees Celsius]
       THEN [wear a jacket]
       ELSE BEGIN
             [wear a short sleeve shirt]
             [bring sunglasses]
       END
```

要想有效地使用伪码，就必须高度重视关键词的使用、字符的缩进以及语句的顺序。

把程序设计问题的解决方案用流程图和伪码清晰地描述出来，是本章介绍流程图和伪码这两种工具的目的。把伪码翻译成汇编语言比直接用汇编语言编程要容易得多。而且，采用流程图或伪码方式设计的程序，便于阅读、调试和维护。

9.2 结构化程序设计的优缺点

结构化程序设计的优点很多，例如：

- 可以方便地跟踪程序的执行，因而有利于程序调试；
- 程序中用到的结构数目有限，且都可用标准化的术语来定义；
- 各个结构可以很容易地转为子例程；
- 结构集是完备的，仅用3种结构，就足以实现所有的程序；
- 各个结构的术语名称就是对该结构的说明，可读性强；
- 可以方便地把各个结构用流程图、语法表、伪码等表示出来；
- 结构化程序设计可以提高编程效率，因为程序编码速度快了。

但是，另外一方面，结构化程序设计也存在一些缺点：

- 仅有少数高级语言(Pascal，C，PL/M)可以直接采用结构化程序设计，其他的计算机语言需要额外做些转换工作；
- 结构化设计的程序，相比非结构化的程序，执行速度较慢，所要求的内存也更多；
- 对某些问题（这类问题数量不是太多）而言，用结构化程序设计，难度反而比较大，还不如直接采用其他曲折复杂的方法；
- 结构如果嵌套，很难弄清其中关系。

9.3 结构化程序设计中的3种结构

用下面3种结构就可以完成所有的程序设计：

- 顺序结构
- 循环结构
- 选择结构

这3种结构看起来不可能构成一个完备集，但如果考虑到结构嵌套（一个结构包含在另外一个结构中），很容易证明：任何程序问题都可以只用这3种结构来实现。下面讨论这3种结构的细节。

9.3.1 顺序结构

顺序结构是程序设计的基本结构。可能在某顺序结构中仅包含1个简单变量赋值指令，例如：

```
[count=0]
```

也可能包括子例程调用：

```
PRINT_STRING ("Select Option:")
```

在源程序的任何地方，只要可以使用单条语句，就可以使用语句组或语句块。若要在伪码中实现顺序结构，只需要把语句用关键字BEGIN和END围起来即可，如下所示：

```
BEGIN
        [statement 1]
        [statement 2]
        [statement 3]
END
```

注意：语句块中的语句要比关键字BEGIN和END缩进几个字符。这是结构程序设计的重要特征。

9.3.2 循环结构

220

第2种基本结构是循环结构。用来重复执行某个操作。把一组数据加起来求和，或在一个列表中搜寻某个值，是两个在程序设计中要用到循环结构的例子。在这两个例子中，循环也可用术语“反复”来替代。尽管循环结构有多种形式，但仅有WHILE/DO和REPEAT/UNTIL这两种形式是结构化设计所必备的。

1. WHILE/DO语句

要实现循环结构，WHILE/DO语句是最容易的方式。由于WHILE/DO作为一个语法单元，只有一个入口，也只有一个出口，和顺序语句看起来差不多，因而被叫作WHILE/DO语句，伪码格式如下：

```
WHILE   [condition]   DO
        [statement]
```

condition是一个关系表达式，求值结果只能是“真”（true）或“假”（false）。如果条件为真，那么执行DO后面的statement语句（或语句块），执行完后重新对conditin关系表达式求值。上述过程不断重复直到条件为假时，跳过DO后的statement语句，继续执行下1个语句。WHILE/DO结构如图9-2所示。

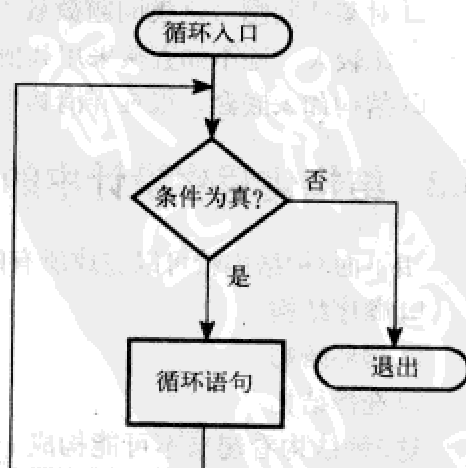


图9-2 WHILE/DO结构流程图

例9-1 WHILE/DO结构

用WHILE/DO结构实现如下功能：只要8051的进位标志被置1，就执行1条语句。

解决方案：

伪码表示如下：

```
WHILE [C==1]  DO
        [statement]
```

注意：= =这里被用来作为关系运算符，测试等式两边的值是否相等，和赋值运算符=(单等于号)不同（见9.4节）。

221

对应的8051汇编代码如下：

```
ENTER:        JNC  EXIT
STATEMENT:    (statement)
              JMP  ENTER
EXIT:         (continue)
```

对应的8051 C语言程序如下：

```
while (C==1)
       {statement;}
```

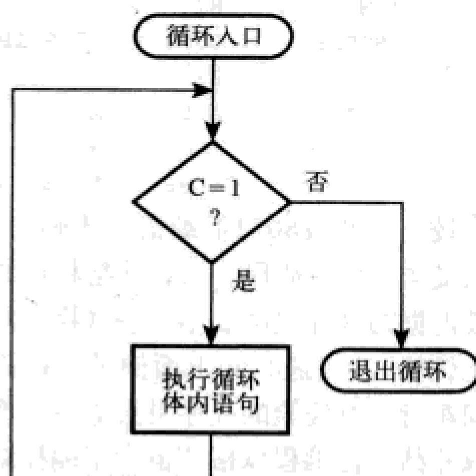


图9-3 例9-1的流程图

一般而言，程序块里的执行语句至少要涉及相关表达式中的一个变量，否则会导致陷入无限循环的错误结果。本例流程图如图9-3所示。

例9-2 求和子例程

写一个名为SUM的8051子例程，用来计算一组数据的和。传递给子例程的参数包括，在R7中存放的数组长度和在R0中存放的数组起始地址（假设数组位于8051内部存储器中）。子例程的返回值存放在累加器A中。

答案：

相应的伪码如下：

```
[sum=0]
WHILE [length > 0] DO BEGIN
              [sum = sum + @pointer)
              [increment pointer]
              [decrement length]
END
```

对应的汇编语言如下：（类结构化，13字节）

```
SUM:        CLR     A
LOOP:       CJNE    R7,#0,STATEMENT
            JMP     EXIT
STATEMENT:  ADD     A, @R0
            INC     R0
            DEC     R7
```

222

```
            JMP     LOOP
EXIT:       RET
```

（非结构化，9字节）

```
SUM:        CLR     A
INC         R7
MORE:       DJNZ    R7,SKIP
            RET
SKIP:       ADD     A, @R0
            INC     R0
            SJMP    MORE
```

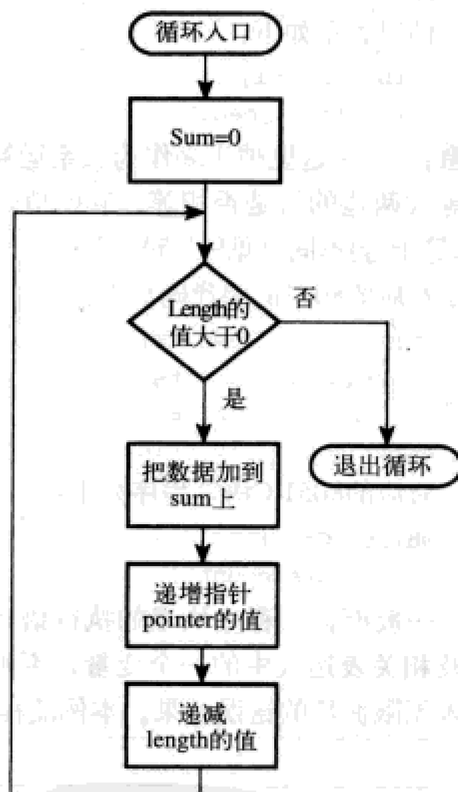


图9-4 例9-2的流程图

比较上面两段8051汇编语言代码，显然，与类结构化代码段相比，非结构化的代码段占据的内存空间要小一些（执行的速度也较快）。当遇到和上面例子一样简单的问题时，有经验的程序员会毫不犹豫地采取非结构化编程风格。但对于编程新手来说，如下的措施是可取的：先把问题的解决方案用伪码清晰地表达出来，然后按照伪码的结构编写汇编语言代码。

对应的8051C语言程序如下所示，本例的流程图如图9-4所示。

```
void sum(int * start, int length)
{
int result = 0 , i = 0;
while (length > 0)
        {
        result = result + start [i];
        i++;
        length- -;
        }
}
```

例9-3 WHILE/DO 结构

编写一个WHILE/DO结构，其中的关系表达式是一个复合表达式：累加器中的数据不等于回车符（0DH）且R7不等于0。

答案：

解决问题伪码如下：

```
WHILE [ACC! =CR AND R7 ! =0] DO
      [statement]
```

请参照流程图，如图9-5所示。

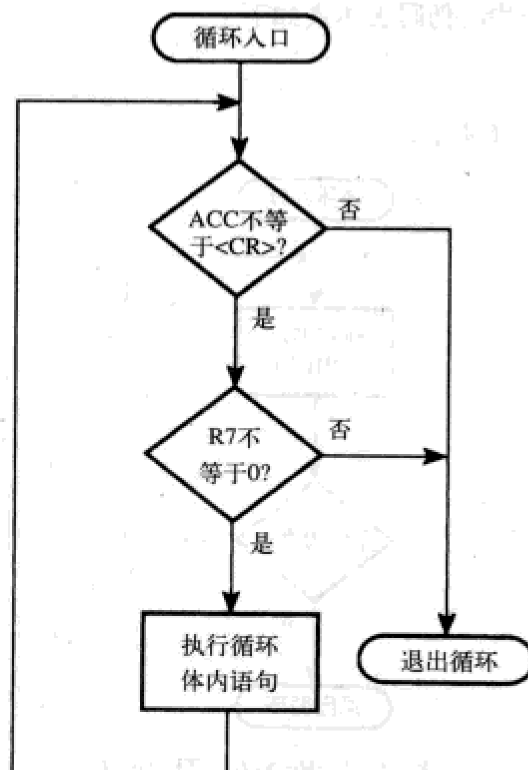


图9-5 例9-3的流程图

相应的8051汇编代码如下：

```
ENTER:          CJNE   A,#0DH, SKIP
                JMP   EXIT
SKIP:           CJNE   R7, #0, STATEMENT
                JMP   EXIT
STATEMENT:      (one or more statements)
                :
                JMP ENTER
EXIT:           (continue)
```

相应的8051 C语言程序如下：

```
while ( (ACC != C) && (R7 ! = 0) )
      {statement;}
```


2. REPEAT/UNTIL语句

REPEAT/UNTIL语句与WHILE/DO语句相似，二者的区别是，REPEAT/UNTIL语句用于循环体内部语句至少要被执行一次的情况。但WHILE/DO语句先测试执行条件是否成立，根据测试结果决定是跳过WHILE/DO结构还是执行WHILE/DO内部

中的语句。因此在多数情况下，WHILE/DO结构的内部语句可能得不到执行。

REPEAT/UNTIL 结构的伪码表达式如下：

```
REPEAT [statement]
UNTIL [condition]
```

对应的流程图如图9-6所示。

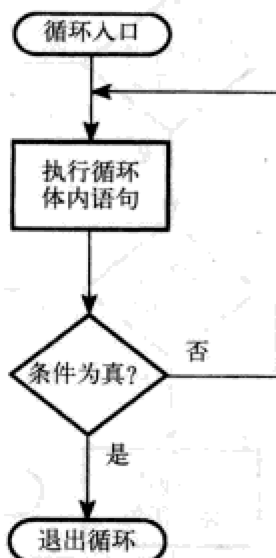


图9-6 REPEAT/UNTIL语句

例9-4 搜索子例程

写一个8051子例程，搜索一个以NULL字符结尾的字符串，该字符串的地址存放在R0寄存器内，如果该字符串包含字母Z，置累加器ACC＝Z，否则，置累加器ACC＝0。

答案：

伪码如下：

```
REPEAT
        [ACC=@pointer ]
        [increment pointer]
UNTIL   [ACC= ='Z' or ACC==0]
```


对应的流程图如图9-7所示。

对应的汇编代码如下：

```
STATEMENT:     MOV    A,@R0
               INC    R0
               JZ     EXIT
               CJNE   A,#'Z',STATEMENT
EXIT:          RET
```

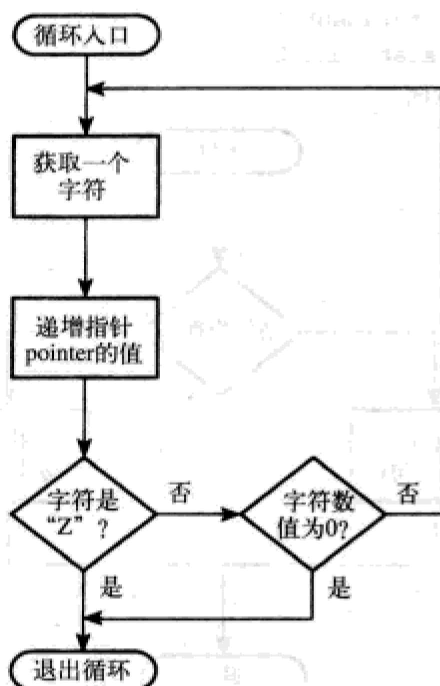


图9-7 例9-4 的流程图

对应的8051 C语言程序如下：

```
char statement(char * start)
{
char a;
do
        {
        a = *start;
        start++;
        }
    while ( (a !='z')|| (a != 0) ) ;
    return a;
    }
```


9.3.3 选择结构

第3种基本结构是选择分支结构，用在程序员需要对程序流进行“分叉”的场合。最常见的两种选择形式为IF/THEN/ELSE语句和CASE语句。

1. IF/THEN/ELSE语句

IF/THEN/ELSE 语句用于根据一定的条件，在两条语句（或语句块）之间必选其一执行的情况。ELSE 部分的语句是可选项。其伪码表达形式如下：

```
IF [condition]
```

```
        THEN [statement 1]
        ELSE [statement 2]
```

对应的流程图如图9-8所示。

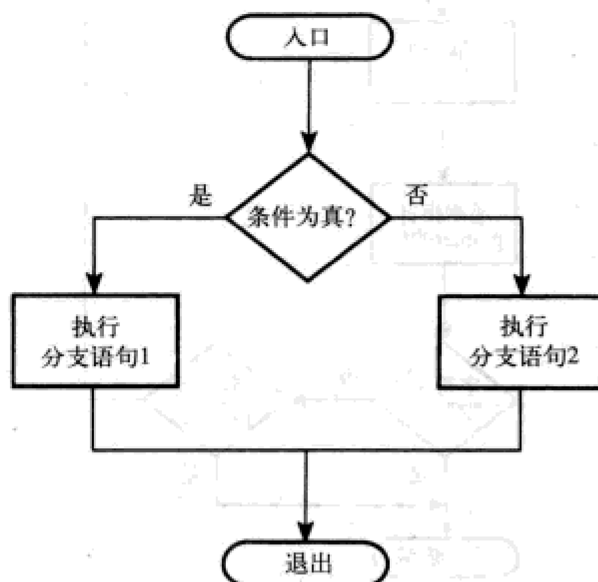


图9-8 IF/THEN/ELSE语句

例9-5 字符测试

写一段指令，测试串行端口输入的字符串，如果是可显示的ASCII字符（20H~ 7EH），输出此字符，否则输出句号“.”。

228

答案：

对应的伪码如下：

```
[input character]
IF    [character = = graphic]
              THEN [echo character]
              ELSE [echo'.']
```

其流程图如图9-9所示。

严格遵循结构化编程风格的汇编代码如下，计14B：

```
ENTER:          ACALL       INCH
                ACALL       ISGRPH
                JNC         STMENT2
STMENT 1:       ACALL       OUTCHR
                JMP         EXIT
STMENT 2:       MOV         A , # '.'
                ACALL       OUTCHR
EXIT:           (continue)
```

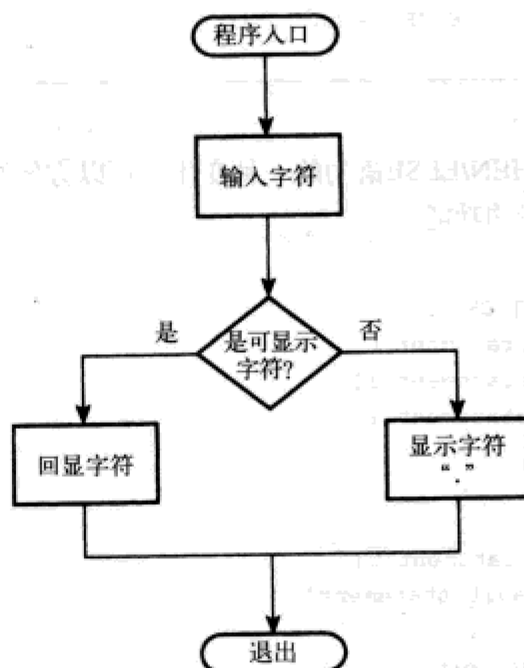


图9-9 例9-5的流程图

不按结构化编程风格编写的汇编代码如下，计10B：

```
        ACALL     INCH
        ACALL     ISGRPH
        JC        SKIP
        MOV       A, #'.'
SKIP:   ACALL     OUTCHR
        (continue)
```

对应的8051 C语言程序如下：

```
while (1)
{
      char a;
      a = inchar ( );
      if (isgrph (a))
            outchr (a);
      else
            outchar ('.') ;
}
```

把上述结构修改后可以反复读入字符：

```
WHILE [1] DO BEGIN
         [input character]
         IF [character == graphic]
              THEN [echo character]
```

```
                    ELSE [echo '.']
      END
```

2. CASE语句

CASE语句是IF/THEN/ELSE语句的一种变体，可以方便地实现根据某个值从多个选项中选择其一执行的功能。

对应的伪码如下：

```
CASE [expression] OF
          0: [statement 0]
          1: [statement 1]
          2: [statement 2]
          .
          .
          .
          n: [statement n]
          [default statement]
END_ CASE
```

230

对应的流程图如图9-10所示。

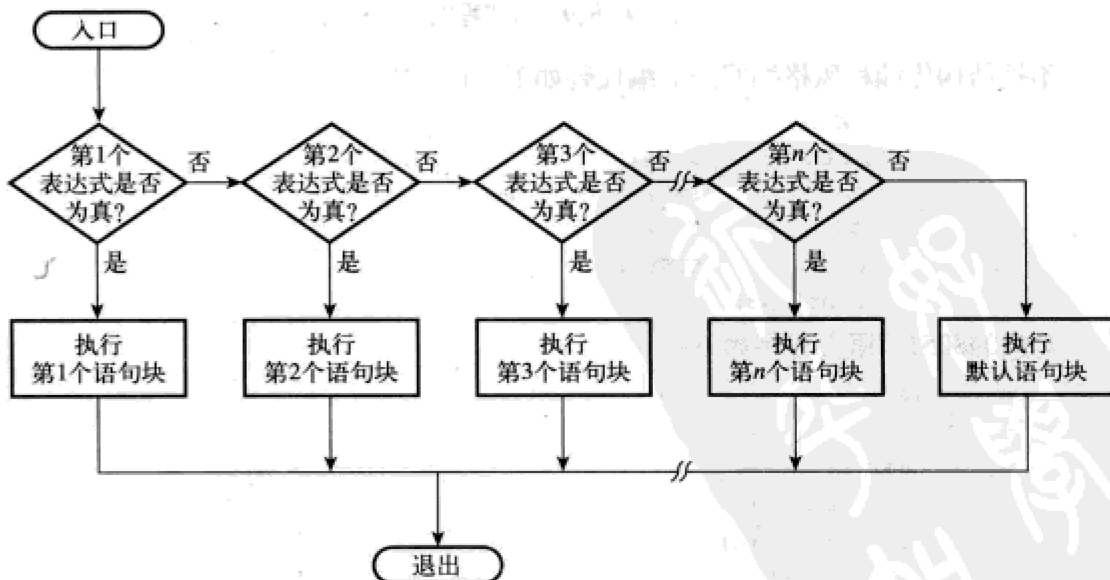


图9-10　CASE结构的流程图

例9-6　根据用户的输入给出相应的回应

在某个菜单回调程序里，要求用户在0、1、2、3中选一，即等效于选择了相应4种操作中的1种。写一段指令，从键盘读入字符，根据输入值，跳到ACT0、ACT1、ACT2、ACT3等处执行程序。（可以忽略错误检查。）

解决方案：

对应的伪码如下：

```
[input a character]
CASE [character] OF
      '0': [statement 0]
      '1': [statement 1]
      '2': [statement 2]
      '3': [statement 3]
END_CASE
```

231

对应的流程图如图9-11所示。

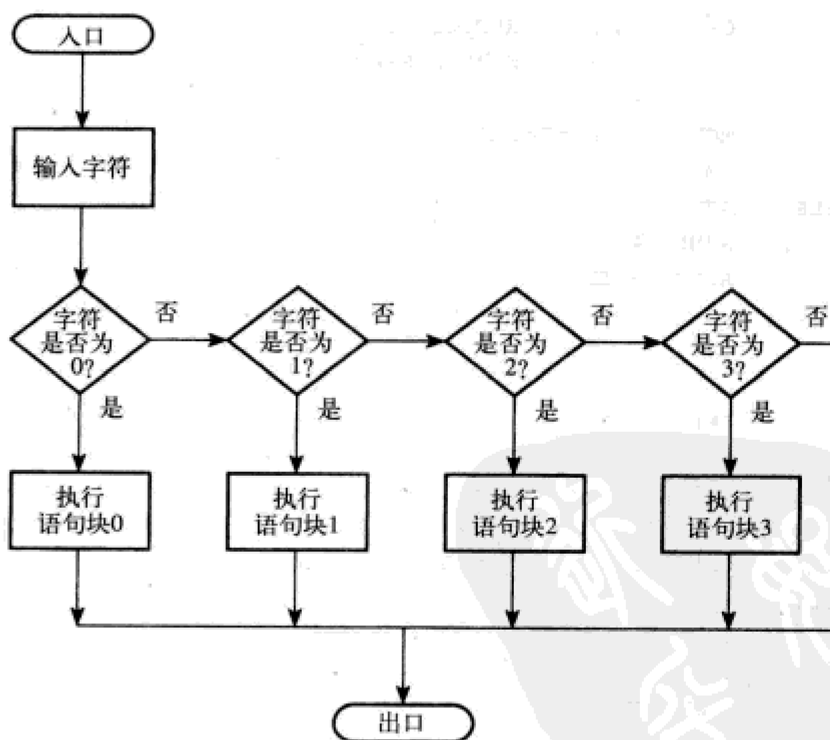


图9-11 例9-6的流程图

严格遵循结构化编程格式的8051汇编代码如下：

```
          CALL INCH
          CJNE A, #'0', SKIP1
ACT0:     .
          .
          .
          JMP EXIT
SKIP1:    CJNEA, #'1', SKIP2
ACT1:     .
          .
          .
```

232

```
            JMP EXIT
SKIP2:      CJNE A, # '2', SKIP3
ACT2:       .
            .
            .
SKIP3:      CJNEA, # '3', EXIT
ACT3:       .
            .
            .
EXIT:       (continue)
```

未按结构化编程的汇编代码如下：

```
            CALL    INCH        ;REDUCE to 2 BITS
            ANL     A,#3        ;WORD OFFSET
            RL      A
            MOV     DPTR,#TABLE
            JMP     @A+DPTR
TABLE:      AJMP    ACT0
            AJMP    ACT1
            AJMP    ACT2
ACT3:       .
            .
            .
            JMP     EXIT
ACT0:       .
            .
            .
            JMP     EXIT
ACT1:       .
            .
            .
            JMP     EXIT
ACT2:       .
            .
            .
EXIT:       (continue)
```

对应的8051 C语言程序：

```
char a;
a = inchar();
switch (a)
{
      case  '0' :
              statement 0;
              break;
      case  '1' :
```

```
                statement 1;
                break;
        case  '2' :
                statement 2;
                break;
        case  '3' :
                 statement 3;
   }
```

3. GOTO语句

组合使用上面介绍的3种结构，总是可以避免使用GOTO语句。但在某些场合，如果程序在某个结构中遇到错误，GOTO语句可以使程序快速地跳出该结构，但使用时需要很小心。在汇编语言程序设计中，和GOTO语句相对应的是无条件跳转指令。在某些情况下，GOTO语句会导致错误。例如，如果8051按照正常的途径进入子例程（采用CALL指令），但离开子例程时没有采用RET指令，而是使用跳转指令离开子例程，那么在调用该子例程时被压入栈的返回地址不会被弹出，依旧保留在栈中，从而最终导致栈溢出。

9.4 伪码语法

考虑到伪码与Pascal和C等高级语言类似，从使用角度来说，给出一个比较正式的伪码定义是必要的。这样，某位程序员用伪码编写的程序可以被另一位程序员读懂并转换为汇编代码。

有一点必须指出，伪码不是在任何情况下都是设计程序的最佳工具。一方面，开发者固然可以方便地在文字处理程序中利用伪码来构建并修改源程序，但另一方面，伪码也和其他程序设计语言一样存在着不足之处：伪码程序是逐行书写的，不能明了地表现程序中的并行动作。流程图则相反，各个并行动作彼此相临，从而提高了相应设计概念模型的清晰度（如图9-10所示）。

在正式介绍伪码的语法之前，先给出一些伪码使用上的小技巧，这些技巧有助于增强采用伪码进行程序设计的能力。

- ❑ 采用描述性的伪码语句。
- ❑ 避免机器依赖性。
- ❑ 用中括号[ ]把条件表达式和语句括起来。
- ❑ 子例程的名字后紧接1对小括号（）。传递给子例程的参数（包括形参和实参）要置于该括号中。
- ❑ 子例程结束处，必须有RETURN，其后紧接1对小括号（）。如果有返回值，要置于括号中。

子例程的例子如下所示：

```
INCHAR ()          OUTCHR (char)        STRLEN (pointer)
    [statement]        [statement]          [statement]
    . . .              . . .                . . .
RETURN (char)      RETURN ()            RETURN (length)
```

❑ 除了保留字和子例程名，其他的代码都使用小写字母编写。

❑ 在结构的入口和出口处，都要采用缩进格式。在循环结构和选择分支结构中，结构内部语句的缩进，比起结构或循环语句关键字所在行的缩进而言，要更进一层。

❑ 采用符号@来表示间接寻址。

推荐采用的伪码语法如下。

保留字：

```
BEGIN      END
REPEAT     UNTIL
WHILE      DO
IF         THEN       ELSE
CASE       OF
RETURN
```

算术运算符：

+	加法运算符
-	减法运算符
*	乘法运算符
/	除法运算符
%	模运算符（即求除法的余数）

关系运算符：（结果为真或假）

==	值相等则为真
!=	值不相等则为真
<	左边值小于右边值则为真
<=	左边值小于等于右边值则为真
>	左边值大于右边值则为真
>=	左边值大于等于右边值则为真
&&	二者都为真则为真
||	二者之一为真则为真

位逻辑运算符：

&	逻辑与
|	逻辑或
^	逻辑异或
~	逻辑非（补码）
>>	逻辑右移
<<	逻辑左移

赋值运算符：

```
=              赋值
op=            赋值运算的速记符号，“op”可以是+、-、*、/、%、<<、>>、 &、^、|中的
               1种。例如，j+ = 4 等价于 j = j+4
```

235

优先运算符：

```
( )
```

间址运算符：

```
@
```

运算符的优先级：

```
( )
~              @
*              /    %
+              -
<<             >>
 <             <=   >       >=
==             !=
&
^
|
&&
||
=              +=   -=      *= etc.
```

注意：

(1) 不要把关系运算符和位逻辑运算符相混淆。位逻辑运算符通常用在赋值语句中，例如：

```
[lower_nibble=byte & 0FH]
```

关系运算符通常用于条件表达式中，例如：

```
IF [char!= 'Q' && char != 0DH] THEN ...
```

(2) 不要把关系运算符“= =”和赋值运算符“=”相混淆，例如，布尔表达式：

```
j == 9
```

根据j的值是否为9，判断表达式的结果是真还是假。而赋值表达式j=9则表示把数值9赋给变量j。

结构如下：

语句：

```
[do something]
```

语句块：

```
BEGIN
```

236

```
      [statement]
      [statement]
      ...
END
```

WHILE/DO：

```
WHILE [condition] DO
       [statement]
```

REPEATE/UNTIL

```
REPEAT
      [statement]
UNTIL [condition]
```

IF/THEN/ELSE：

```
IF  [condition]
        THEN [statement 1]
        (ELSE [statement 2])
```

CASE/OF：

```
    CASE [expression] OF
         1: [statement 1]
         2: [statement 2]
         3: [statement 3]
         .
         .
         .
         n: [statement n]
         [default statement]
    END_CASE
```

9.5　汇编语言编程风格

在进行汇编语言程序设计时采用一种清晰连贯的编程风格是很重要的。如果程序员在一个团队里工作，这一点尤为重要，因为每个成员都需阅读和理解其他成员编写的源代码。

到目前为止，对采用汇编语言编程来解决实际问题，本书的相关介绍还不是很深入，因此对编程风格的介绍也就很粗略。然而，在程序设计规模比较大的时候，对程序设计风格的要求，相比前面介绍的内容，要严格得多。下面是一些关于程序设计风格的小建议，可以帮助开发者改进汇编语言程序设计风格。

9.5.1　标号

标号代表地址，标号名应该能够描述其所代表的目的地址。例如，在重复执行某个操作的场合，可以用诸如LOOP、BACK、MORE等标号代表跳转回去的目的

地址；如果要跳过程序中的一些指令，可以用标号SKIP或AHEAD来表示目的地址；如果要反复检查一个状态标志，指令中可以用标号WAIT或AGAIN表示目的地址。

237

如果汇编器是常驻内存型的，或是采用绝对地址的汇编器，那么，在标号的使用上会受到一些限制。因为这类汇编器把整个程序作为一个待汇编的单元来处理，从而限制了一些通用标号的使用。有几个办法可以解决该问题。例如，可以把通用标号按序编号，使用诸如SKIP1、SKIP2、SKIP3这样的标号，以示区别。又如，在子例程内，所有标号的命名按照如下规则：子例程名＋数字，标号形如SEND1、SEND2、SEND3等。但这样的话，程序看上去就明显不清晰了，因为标号SEND2和SEND3无法形象地指示当前指令是“跨越”操作，还是“循环”操作。

对于ASM51这样性能强大的汇编器来说，允许不同的子例程（或一组同类型的子例程）存在于不同的文件之中，并可独立于主程序对该文件单独进行编译。主程序在自己的文件内被汇编，可用链接/定位器解决各个目标文件间的外部符号引用问题，并把主程序和各个子例程组合起来，形成最终的目标程序。这种类型的汇编器，常常被称作“可重定位”的汇编器，允许同一标号出现在不同的文件中。

9.5.2 注释

在程序设计中无论怎样强调注释也不过分，尤其在使用汇编语言进行程序设计的时候，由于其固有的抽象特性，所以更要重视注释的作用。所有的代码行，除非代码本身的含义非常明确，都要添加注释。

对于条件跳转指令，一种比较有效的注释方式，类似于流程图上跳转操作的“问题”式注释。并将回答yes或no放在注释里，分别代表jump和no jump操作。下面的INLINE子例程里带有一些注释，注意观察测试字符是否为回车符<CR>的代码行的注释风格。

```
;**********************************************************************
;
;INLINE          INPUT LINE OF CHARACTERS
;                LINE MUST END WITH<CR>
;                MAXIMUM LENGTH 31 CHARACTERS INCLUDE<CR>
;ENTER:          NO CONDITIONS
;EXIT:           ASCII CODES IN INTERNAL DATA RAM
;                0 STORED AT END OF LINE
;USES            INCHAR,OUTCHR
;
;**********************************************************************
;
INLINE:          PUSH   00H          ;SAVE R0 ON STACK
                 PUSH   07H          ;SAVE R7 ON STACK
```

238

```
                PUSH  ACC           ;SAVE ACCUMULATOR ON STACK
                MOV   R0,#60H       ;SET UP BUFFER AT 60H
                MOV   R7,#31        ;MAXIMUM LENGTH OF LINE
STMENT:         ACALLINCHAR         ;INPUT A CHARACTER
                ACALLOUTCHR         ;ECHO TO CONSOLE
                MOV   @R0, A        ;STORE IN BUFFER
                INC   R0            ;INCREMENT BUFFER POINTER
                DEC   R7            ;DECREMENT LENGTH COUNTER
                CJNE  A,#0DH,SKIP   ;IS CHARACTER=<CR>?
                SJMP  EXIT          ;YES: EXIT
SKIP:           CJNE  R7,#0,STMENT  ;NO: GET ANOTHER CHARACTER
EXIT:           MOV   @R0,#0
                POP   ACC           ;RETRIEVE REGISTERS FROM
                                    ;STACK
                POP   07H
                POP   00H
                RET
```

9.5.3 注释块

在每个子例程源代码的起始处添加若干行的注释是非常必要的。由于子例程所完成的任务都是经过精心定义的，子例程的功能必须具备通用性，且具有详细的说明文档。从这个角度来说，每个子例程前都应该有个注释块，并在注释块中说明如下内容：

- 子例程名
- 操作功能
- 调用条件
- 退出条件
- 如果在该子例程中要调用其他子例程，给出其名称。
- 列出该子例程可能影响到的寄存器的名称。

上面的INLINE子例程是一个注释完备的好例子。

9.5.4 在栈中保存寄存器

随着应用程序的规模和复杂度的增长，开发者在新编写一个子例程时，常常要用到已经存在的子例程。因此，子例程会调用另外的子例程，而另外的子例程也会调用其他的子例程，这种现象被称为“子例程的嵌套”。只要栈有足够大的空间来存放返回地址，“子例程嵌套”就不会出现任何问题。当然在实际编程时，多层子例程嵌套的情况很少见。

然而，在子例程里使用寄存器会导致1个潜在的问题。随着子例程嵌套层次的增加，要跟踪哪些寄存器受到子例程调用的影响就会越来越困难。惯用的做法是，

在子例程入口处，把那些受子例程影响的寄存器预先保存在栈上，并在子例程结束时，从栈恢复这些寄存器的值。请注意，上面例子中的INLINE子例程里，就使用了栈来保存和恢复R0、R7以及累加器等寄存器，在INLINE子例程返回到被调用点后，R0、R7以及累加器的值和调用INLINE子例程之前相同。

9.5.5 等于语句的作用

用等于语句定义常量可以使程序易于阅读和维护。用于程序的开头，定义诸如可以定义代表回车符号 (CR) 和换行符号 (LF) 的常量符号，也可以定义如STATUS、CONTROL这类代表外围集成电路中的某个寄存器地址的常量符号。

定义常量符号后，在整个程序里，都可以用常量符号来替换其所对应的数值。在汇编的时候，汇编器负责把常量符号转换为相应的数值。大量使用常量可以使得程序维护起来更方便，可读性更好。如果需要改变常量，只需要改变定义常量符号的那行源代码，重新汇编程序后，程序中凡是用到该常量符号的地方，对应的数值都会被更新。

9.5.6 子例程的使用

如果程序规模比较大，就有必要采用“分而治之”的编程策略。换句话说，将大而复杂的任务划分为若干个小而简单的任务。程序设计中，这些小而简单的任务可以通过子例程的形式来完成。子例程有其层次性，也就是说，简单的子例程可以被比较复杂的子例程所调用，而较复杂的子例程又可以被更复杂的子例程所调用，以此类推。

在流程图中用“预定义过程框”来调用子例程（如图9-1所示）。这表明，另有一流程图来说明该子例程的工作细节。

在用伪码写子例程时，子例程的代码结构必须完整，开头必须是子例程名和括号，括号内是传递给子例程的参数名或数值（如果有）。各个子例程以关键字RETURN结尾，其后紧跟一对括号，括号内是子例程的返回值（如果有）。

如果要举一个关于子例程层次关系的例子，最简单的也许就是输出字符串的子例程OUTSTR和输出字符的子例程OUTCHR。OUTSTR作为高层子例程调用处于低层的子例程OUTCHR。

流程图、伪码、8051汇编语言和C语言的解决方案如下所示。

伪码：

```
OUTCHR (char)
    [put odd parity in bit 7]
    REPEAT [test transmit buffer]
    UNTIL [buffer empty]
    [clear transmit buffer empty flag]
```

```
        [move char to transmit buffer]
        [clear parity bit]
RETURN ()
OUTSTR (pointer)
        WHILE [ (char = @pointer) !=0] BEGIN
                OUTCHR (char)
              [increment pointer]
        END
RETURN ()
```

240

汇编语言：

```
OUTCHR:     MOV     C, P            ;PUT PARITY BIT IN C FLAG
            CPL     C               ;CHANGE TO ODD PARITY
            MOV     ACC. 7, C       ;ADD TO CHARACTER
AGAIN:      JNB     TI, AGAIN       ;TX EMPTY?
            CLR     TI              ;YES: CLEAR FLAG AND
            MOV     SBUF, A         ;SEND CHARACTER
            CLR     ACC .7          ;STRIP OFF PARITY BIT AND
            RET                     ; RETURN
OUTSTR:     MOV     A, @DPTR        ;GET CHARACTER
            JZ      EXIT            ;IF 0, DONE
            CALL    OUTCHR          ;OTHERWISE SEND IT
            INC     DPTR            ;INCREMENT POINTER
            SJMP    OUTSTR          ;AND GET NEXT CHARACTER
EXIT:       RET
```

流程图如图9-12和图9-13所示。

8051 C语言：

```
sbit AccMSB = ACC^7;
void outchr (char a)
{
ACC = a;
CY = P;
CY = ! CY;
AccMSB = CY;
while (TI ! = 1);
TI = 0;
SBUF = ACC;
AccMSB = 0;
}
void outstr(char * msg)
}
while (*msg !='\0')
     outchr (*s++);
}
```

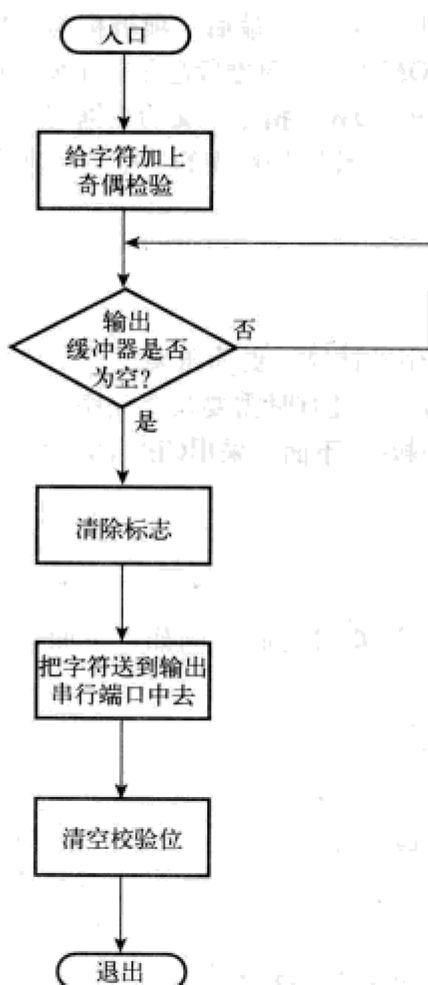


图9-12 OUTCHR子例程流程图

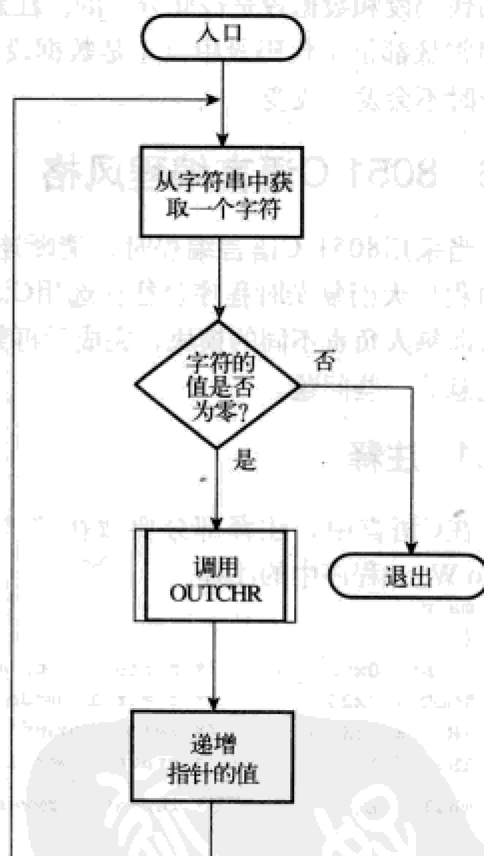


图9-13 OUTSTR子例程流程图

241
~
242

9.5.7 程序的结构

开发者往往采用模块化的方式来设计程序(例如,子例程独立于主程序被编写),但各个源程序模块应当在布局上彼此相连贯。一般来说,源程序各部分模块的布局如下:

- 常量定义
- 初始化指令
- 程序主体
- 子例程
- 数据常数定义(DB和DW)
- 在RAM中定义的数据块(DS伪指令)

上述各项，除了最后一项外，被统称为“代码段”，而最后一项被称为“数据段”。习惯上，由于代码段常常位于ROM或EPROM中，而数据段总是位于RAM中，因而代码段和数据段是彼此分开的。注意，用DB和DW伪指令定义的数据常数和字符串常量都位于代码段中（不是数据段），因为这些数据是程序的一部分，在程序执行时不会发生改变。

9.6 8051 C语言编程风格

当采用8051 C语言编程时，清晰连贯的程序设计风格更为重要。这是因为当8051程序大而复杂时程序员往往选用C语言编写，并且有时需要几个程序员协同工作（即每人负责不同的模块，完成后再整合到一起）。下面是采用C语言编辑8051时要注意的一些问题。

9.6.1 注释

在C语言中，注释部分要放在“/*”和“*/”符号之间。例如，下面所示的Hello World程序中的注释。

```
main ()
{
SCON = 0x52;          /* serial port, mode 1 */
TMOD = 0x20;          /* timer 1, mode 2 */
TH1  = -13;           /* reload count for 2400 baud */
TR1  = 1;             /* start timer 1 */

while (1)             /* repeat forever */
      {
      printf ("Hello World\n"); /* Display "Hello World" */
      }
}
```

243

9.6.2 定义的使用

采用常数名定义常数会使程序更具可读性，而且需要改变常数时只要简单地改变1行代码就可以了。如下例所示。

```
#define PI 3.1415927
#define MAX 10

int circleArea (int radius)
{
return (PI * radius * radius) ;
}

main ()
{
int i;
for (i = 0; i < MAX; i++)
```

```
  circleArea (i) ;
}
```

上面提到的程序将PI定义为常数π=3.1415927，同时MAX定义为常数10。PI被用于计算一个圆的面积，而MAX作为可计算的不同面积圆的最大个数。例如，若想计算20个不同圆的面积，那么MAX的值只需简单地将定义行改变为#define MAX 20。

9.6.3 函数的使用

大而复杂的C语言程序通常是由几个程序员同时设计编辑的。这可能是通过模块化程序设计方法来实现的，即将程序划分成一些模块和函数。将程序分化成函数，允许更多的模块化而且使整个程序更加清晰和容易阅读。该方法在第12章中用于接口和设计程序实例中。

9.6.4 数组和指针的使用

当需要存储一个有序的相关数据时，最好的方法是使用数组并通过指针指向这个数组。调用的指针存储了存储器的位置地址。因此，数组第一个元素的地址被赋予一个指针，那么每个数组元素可通过在指针值上加适当的偏移量进行间接存取。例如，通过使用数组，查表将得到最优化执行。

数组对字符串存储是非常有用的。例如，This is a welcome message这条信息可通过一个数组存储到存储器，这仅需简单的1行命令：char *MSG = { "This is a welcome message. " }，即使用一个有序的字符来存储字符串。

244

9.6.5 程序结构

为了布局和结构的标准化，我们在编写8051 C语言程序时采用下列程序结构：

- ❑ 库函数引用
- ❑ 常数定义
- ❑ 变量定义
- ❑ 函数类型定义
- ❑ 主函数
- ❑ 函数定义

常数定义是如#define PI 3.1415927这样的语句，变量定义是类似char A语句。函数类型定义是像void HTOA (void) 这样的1行指令，它由函数名、返回类型和参数清单组成。而函数定义是对函数所包括的所有操作进行定义。如下例所示：

```
void HTOA (void)
{
A=A&0xF;
if (A>=0xA)
```

```
        A = A+7;
    A=A+'0';
}
```

小结

本章介绍了流程图和伪码，并以此讨论了结构化程序设计技术。此外，本章还给出了若干在设计和展示程序时增强程序可读性的建议。在下一章将会介绍一些用于程序开发的工具和技术。

习题

9.1 采用WHILE/DO结构实现，当累加器的值小于或等于7EH时，执行某个语句块。

9.2 写出一个用于WHILE/DO结构的复合条件表达式：累加器大于0，R7大于0且进位符号的值为10，假设累加器和R7中的数据是无符号数。

9.3 写一个子例程，在累加器中数据的各个“位”里，找出权重最大的1的位置，在R7中返回该位置的值。例如，如果ACC＝00010000B，那么R7＝4。

9.4 写一个名为INLINE的子例程，从控制台输入一行字符串，把输入的字符串放在内部RAM中起始地址为60H的数据单元中。该字符串的最大长度为31个字符（包括回车符号）。在字符串的结尾处是0。

245~246

第10章 用于程序开发的工具和技术

10.1 引言

本章介绍了若干不同类型的开发工具，并分步骤介绍开发微控制器的产品（或基于微处理器的产品）的相关流程。要把一个设计概念转化为对应的产品，中间需要经过很多的流程，而且要用到大量的开发工具。本章介绍基于8051微控制器的典型设计中最常用的开发流程和工具。

无论是个人开发者还是开发团队，都必须了解一点：设计是一项要求高创造性的脑力活动，应当在开发的计划阶段留出一定的设计余量，便于在项目进行过程中做适当的改变。如果程序规模很大，或者项目对安全性要求很高，要自觉做到这点可能会比较难。但无可否认，在这些情况下，必须对项目的流程管理和验证提出更高的要求。本章关注的是那些规模相对较小的产品，例如用于如下产品的控制器：微波炉、汽车仪表板、计算机外设、电动打字机以及高保真音响。

本章将讨论开发设计中必经的几个步骤、使用的工具和技术，并给出若干设计案例。了解开发编程很重要，但不鼓励读者严格按照书本上介绍的流程进行。这是因为，业界认识到，如果强制开发者遵循孤立的、顺序既定的开发流程，那么有可能工作量偏大或导致错误。本章的后面会介绍一个包含所有开发流程的例子，例子中用到的各种技术都是现成的，开发者可以自行决定如何开发。下面先介绍开发周期的各个流程。

10.2 开发周期

通常用流程图来描述从概念到产品的各个开发步骤，整个过程被称为**开发周期**，如图10-1所示。读者可能会注意到，各个步骤之间并没有特别显示出任何“周期”的含义。实际上，图10-1所示的开发流程是一个理想的开发过程，整个过程没有考虑开发中可能会出现的“故障”，当然在实际运作中，总会遇到一些问题。如果开发周期的任何一个步骤出现问题，都得进行调试（找出并修正错误），并对早期的设计做出相应的修改。问题的严重程度不同，修改的幅度也就不同，最极端的情况是，开发者可能需要回到概念阶段从头开始。因而，也就暗示着图10-1所展示的开发周期中的任意一个步骤的输出结果，都可能和之前的任何一个步骤有关。

图10-1上半部分所示的各个步骤对应软件开发过程，下半部分所示的各个步骤

对应硬件开发过程。二者相结合的步骤是"整合与验证"，这个步骤很关键，也很复杂，决定着"产品"设计的成败。如果要做成面向最终用户的产品，还需要几个步骤，在图中没有描述进来，例如，制造、测试、产品发布和市场等。图10-1中虚线框内的各个步骤是本章重点内容，实际上也是本书的研究重点。后面会更详细地讨论这些流程，但下面先简略看看软件开发中的各个步骤。

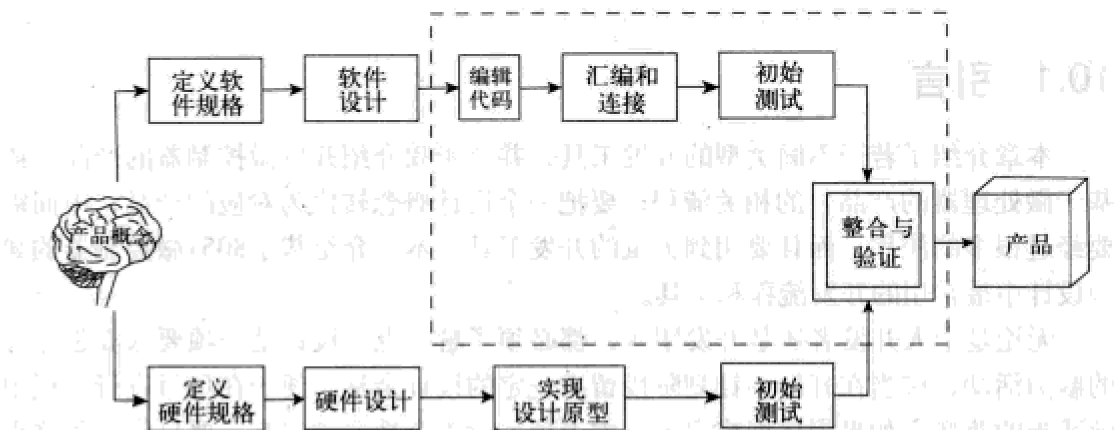


图10-1　开发周期示意图

10.2.1　软件开发

本节讨论图10-1上半部分所示的各个流程，先讨论如何定义应用软件的规格。

定义软件规格。定义软件的规格，就是要明确说明该软件要做什么。定义软件规格有好几种方法，在低层次上，可以先给出用户界面的定义，也就是说，用户怎样和系统交互并控制系统（用户的每个行动将导致什么样的结果）。如果硬件原型上用到了开关、刻度盘，或者音频和视频指示器，应当明确地定义各个部件的用途和操作规程。

248

计算机领域的科学家为定义软件需求而设计了一套正规的方法，但在基于微控制器的应用开发中，通常不会用到这些方法。因为比起大型计算机上的应用程序而言，微控制器上运行的程序规模要小一些。

软件定义也可能包括用户界面下的系统操作的细节。例如，为了确保复印正常安全地进行，复印机的控制器必须对复印机内部诸如温度、电流、电压以及送纸状况等各种参数进行监控。这些参数和用户界面没有什么大关系，但设计软件时必须考虑这些参数。

软件定义可以模块化，各个系统函数带有入口和出口条件，用于模块间的通信。显然，前面章节子例程部分介绍的技术，在软件定义的第一步就会用到。

如果一个系统是中断驱动的，相应的软件定义需要仔细规划，而且在软件定义阶段必须考虑一些特殊的细节。对实时性要求不是太高的任务可以放在前台循环中

执行，或是放在由定时器中断处理的轮询序列中执行。实时性要求高的任务，需要将其对应的中断优先级设置为高，当中断产生时，系统可即时处理该任务。这样的系统，在软件定义时，必须强调对执行时间的要求。例如，每个子例程和中断服务程序（Interrupt Service Routine，ISR）的执行时间是多少？每个ISR多长时间执行一次？处理异步任务（用于响应事件）的ISR可能随时需要系统进行处理，因此，在某些进程中有必要阻塞此任务，而在另外一些进程中，需让该任务（中断驱动）运行。对这样的系统，软件定义要考虑各个任务的优先级，轮询的先后次序，如果需要的话，还得考虑如何在ISR内部动态调整优先级，或者动态改变轮询次序。

软件设计。软件设计是需要开发过程的一个不可缺少的步骤，但开发人员往往没有做太多计划，就直接着手进行软件设计。在此阶段，在编写汇编代码之前，应该先用流程图和/或伪码将设计任务的解决方案描述出来，也就是在正式写代码前做好计划。第9章已经对此进行了详细的讨论。

编码和汇编。在软件开发过程中，编码和汇编这两个流程可能会重复进行多次，即使再紧凑的开发周期，至少在软件开发的开始阶段是这样的。如果汇编器在汇编时发现代码中的错误，就需要重新修改源代码，之后再次进行汇编。由于汇编器在汇编的时候，不可能对程序的目的有所了解，因而只能检测出源代码语法方面的错误（比如，代码行中少了1个逗号，或使用了未定义指令等），这类错误被称为**语法错误**，也叫作**汇编错误**。

初始测试。运行时错误要到程序在仿真器上或目标系统上运行时才能被发现。这类错误可能很隐蔽，要想找出这类错误，需要在程序执行时对CPU在各个阶段的状况进行仔细观察。**调试器**是一个可以执行用户程序并帮助用户寻找“运行错误”的系统程序。调试器具有一些特性，例如，执行用户程序直到某个特定的地址（**断点**）时暂停执行。调试器还可以单步执行程序中的各条指令，并同时显示CPU当前的各个寄存器、状态位以及输入输出端口的状况。

10.2.2 硬件开发

本书前文极少涉及硬件开发。这是因为8051微控制器是一个集成度很高的器件，本书聚焦于8051内部结构的学习以及通过软件来开发片上各种资源。到目前为止，本书中介绍的各个例子仅仅是使用了一些简单的接口和外部器件相连接。

249

定义硬件的规格。定义硬件的规格，意味着给出系统函数所需的定量数据。例如，开发一个机械式手，需要定量说明如下参数：铰链的个数、伸展距离、速度、精确度，转矩以及功耗等。通常会要求硬件设计人员提供一张类似音频放大器或VCR说明书的规格清单。除此之外，硬件规格还包括其他一些参数：产品的尺寸和重量、CPU的运算速度、内存的类型和容量、内存映射和分配、I/O端口以及可选项等。

硬件设计。传统的硬件设计方法要用到铅笔和逻辑模板，该设计方式至今仍然被广泛使用，但借助计算机辅助设计 (Computer-Aided Design, CAD) 软件，可以提高该方法的设计效率。尽管很多CAD工具软件原本是用于机械和民用工程领域的，但其中部分CAD软件可用于电子工程领域。最常见的两类CAD工具，一类用于绘制电路原理图，另一类用于绘制印制电路板图 (Printed Circuits Board，PCB)。学会如何使用这类工具需要花费不少时间，但一旦掌握后，设计效率会大幅提高。某些画原理图的程序能够以文件的形式输出结果，PCB工具软件读取这些文件之后，自动完成印制电路板的布线工作。

构建设计原型。要实现一个硬件系统的原型，必须付出大量劳动，此外别无它法。不管是在面包板上完成某个单板机的一个简单的总线接口或连接器接口，还是在一个安稳的控制板上绕接，都需经过大量的实践，才能开发出相应硬件系统的原型来。大公司资金充足，可以直接以印制电路板的形式来实现硬件系统的原型，甚至第一次试验就能采用PCB来进行。小公司、学生或者电子技术爱好者从事的项目一般较小，要实现硬件原型，更倾向于采用在面包板上布线的传统方法。

初始测试。第一次测试硬件不会用到任何应用软件程序。硬件测试必须逐步推进，原因很简单，如果还没有验证硬件的供电电压，用示波器观察时钟信号又有何用呢？下面是一个可行的测试步骤：

- 目测
- 检查连线
- 直流信号测试
- 交流信号测试

在面包板硬件系统上电之前，必须进行目测和连线检查。检查连线时，用欧姆表对各个IC进行引脚到引脚的检测。对“引脚到插座”和“插座引脚到连线”等处的连通性进行验证。在硬件首次上电前，要将各个IC芯片从插座上取下来。上电后，要用电压表在面包板的各处测试直流电压是否正确。上述检测都通过后，把IC芯片插上去，最后再测试交流工作情况，验证时钟信号等。

在验证了线路连接、电压供应和时钟信号后，接下来就进入硬件调试的实质性阶段了。硬件原型的功能是否与设计的预想一致？如果不是，设计者可能得回到前面步骤，重新构建硬件原型，或者重新设计硬件系统，或者重新定义硬件的规格。

如果系统中采用CPU，那么一个简单的“写错误”就可能导致CPU的初始复位过程无法完成，因为CPU无法执行复位后的第1条指令。对此类错误，一个行之有效的调试方法如下：把1个低频方波（频率为1kHz）加到CPU的复位信号线上，用示波器或者逻辑分析仪观察CPU复位后总线上的响应情况。

250

硬件系统的功能测试需要应用软件或监控软件的配合和帮助才能完成。

10.3 整合和验证

在产品开发周期中，软硬件联合调试是最困难的一个阶段。某些非常隐蔽的错误，在仿真（如果进行了仿真）的时候没有检测出来，但在实时执行的时候却冒了出来。要发现此类错误，需要综合利用各类资源：硬件方面，要用到PC机开发系统、目标系统、电源、电缆以及测试设备；软件方面，要用到监控程序、操作系统、终端仿真程序等。

后面会详细讨论整合和验证这一流程，但现在先把图10-1中用虚线围起来的各个流程扩展成如图10-2所示框图。

图10-2中用圆圈代表开发中用到的实用程序和开发工具，用方框代表用户文件，用双划线方框来代表“执行环境”。图10-2中，第1步就是用编辑器来编写源程序。但图10-1中的转换（translation）流程被细分为两个步骤。先用汇编器（比如ASM51）把源程序文件转换为目标文件，再用链接/定位器（比如RL51）把各个可重新定位的文件组合起来，生成1个可在目标系统或仿真器上执行的，采用绝对地址的目标文件，汇编器和链接/定位器还生成相应的列表文件。

图10-2中括号内是各种类型文件最常用的后缀名。任何形式的文件名和后缀都可以作为调用参数传递给汇编器，但不同种类的汇编器，默认值（文件名和后缀名）会不同。

如果程序的源代码都包含在一个文件中，且汇编时生成的是采用绝对地址的目标文件，那么就不需要链接和定位操作了。考虑到这种情况，图10-2中在常用流程之外，另外给出了1条路线，表明汇编器可以直接生成采用绝对地址的目标文件。

开发者也可以采用诸如C、PL/M这类的高级语言来代替汇编语言（虽然本书中没有特别强调），或者和汇编语言一起开发应用软件。在转换的时候，就需要有一个**交叉编译器**来生成供链接和定位的可重新定位的目标模块。

在转换时，可能要用到类似Intel LIB51这样的**库**。一些可重新定位的目标模块，通用性很强，可以用于很多软件项目中（绝大部分时候，以子例程的形式），完全可以把这些模块以“库”的形式存放在一起。库名作为调用参数传递给RL51，RL51在连接和定位各个目标模块时，如果某些模块中声明的外部符号（子例程）无法在源代码中找到相应定义，RL51会搜索“库”，查找是否有匹配的代码段。

10.3.1 软件仿真

图10-2中给出了5个执行环境。图10-1中的初始测试是在没有目标系统的情况下执行的。在图10-2中，把这称为软件仿真。**仿真器**在开发系统的主机上运行，模拟目标机器的结构。例如，8051的仿真器包括和8051各个特殊功能寄存器相对应的虚拟（模拟）寄存器，和8051内部/外部内存相对应的虚拟内存。8051程序在仿真模

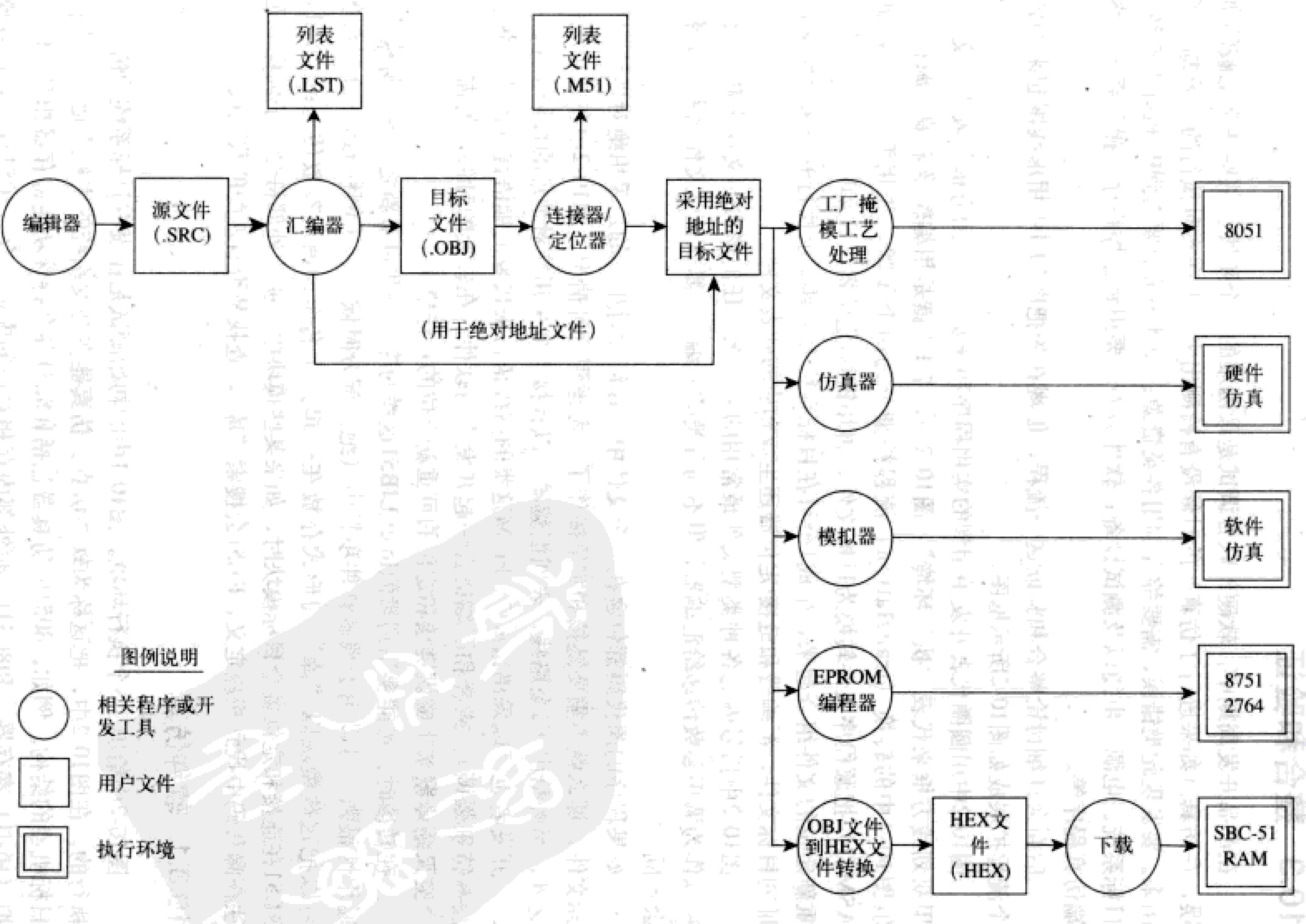


图10-2 开发周期中的详细步骤

式下执行，结果显示在开发主机的CRT显示器上。仿真器在早期测试中很有用，但程序中直接牵涉到硬件那部分代码，必须运行在目标机器上才能进行测试。

10.3.2 硬件仿真

通过**硬件仿真器**（联机仿真器），可以直接把开发系统和目标系统联系起来。仿真器包含一个处理器，该处理器可以替换目标系统上的处理器。但仿真器内的处理器可被开发系统直接控制。因而，软件在目标系统上执行的时候，开发系统能够对软件加以控制，可在开发系统上输入命令，对目标系统上的软件进行单步调试，还可以通过开发系统，在软件中设置断点（单个或多个）。而且，那些在时序严格情况下才可能出现的错误，无法在软件模拟执行时被发现，但应用硬件仿真器后，软件在目标系统上全速执行，因而可以发现此类的错误。

价格昂贵是硬件仿真器的最大不足之处，基于PC机的开发系统的售价在2000美元到7000美元之间。大多数电子爱好者无法承受这样的价格，这个价格也超出了大多数大学的预算范围（如果要给整个实验室装备这样的系统）。但那些需要一个专业开发环境的公司，会毫不犹豫地在硬件仿真器上加以投资，从而加快产品的开发，所获得的收益足以弥补购买硬件仿真器的开销。

10.3.3 在RAM中执行程序

除了以上介绍的，还有一种简单有效的方法，无需用到硬件仿真器就可在目标系统上运行待测的软件。倘若目标系统上包括外部RAM，且设计时使得该RAM在内存空间里和外部代码空间重叠（使用第2章里介绍的方法，请参考2.6.4节），那么采用绝对地址的目标文件，可以从开发系统的主机上，下载到目标系统上，并在目标系统上运行。

Intel十六进制格式：如图10-2所示，若要把目标文件下载到带有RAM的目标系统上执行，为了传输方便，需要一个额外的翻译步骤，把目标文件转换为标准的ASCII码格式。目标文件里包含的是二进制代码，无法打印或者显示。但采用十六进制数可以克服这个缺点。例如，字节1AH对应的ASCII码是1个控制字符，不是1个可以输出显示的字符，因而无法传送给1台打印机。但打印机可以接受字节31H（“1A”中字符“1”对应的ASII码）和41H（“1A”中字符“A”对应的ASII码），因为它们对应的ASCII码可以输出显示。实际上，字节31H和41H对应的ASCII码打印出来后，就是“lAH”（请参考附录F）。

Intel十六进制格式是一种把机器语言二进制程序保存为ASCII字符文件（可读可显示）的标准。Intel十六进制文件包含许多行字符（每行都可被看为1个记录），记录包含如下字段：

252
≀
253

字段名	字节数	内容说明
记录标志（Record mark）	1	“:”指示接下来是1个记录
记录长度（Record length）	2	1个记录里面的数据字节数
载入地址（Load address）	4	记录所包含的数据在内存中的起始地址
记录类型（Record type）	2	00代码数据类型的记录，01表示记录结尾
数据（Data bytes）	0~16	记录中存放的数据
校验和（Checksum）	2	把记录中前面所有字节加起来的和加上Checksurm等于0

图10-3是一个Intel十六进制格式文件，图中给出了记录中各个字段的相应标注。转换程序读入采用绝对地址的目标程序，把机器语言的二进制字节转为如上所述的Intel十六制格式，最后生成一个Intel十六进制格式的文件。Intel公司提供的转换程序名为OH。

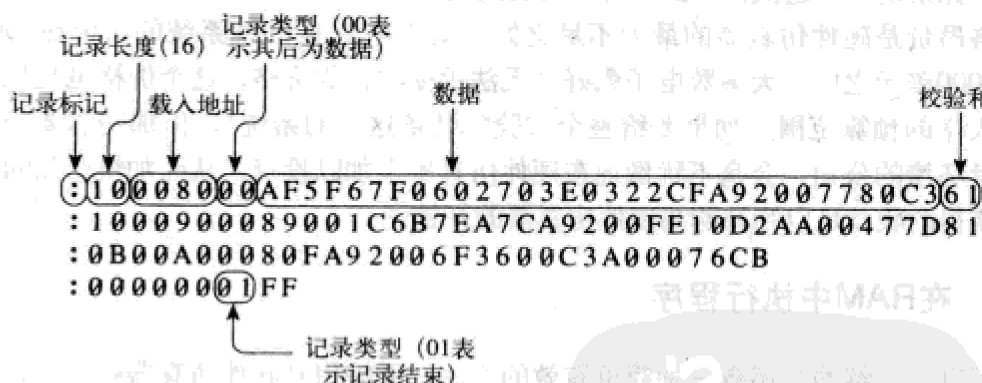


图10-3　Intel十六进制文件的格式

10.3.4　在EPROM中执行程序

程序先在RAM中（或在仿真器中）测试执行，如果性能达到了预期的满意度，那么就要把程序代码烧录到EPROM里，然后作为**固件**安装到系统里。图10-2中举了两种类型的EPROM作为例子。8751代表8051的EPROM版本，2764则是一种在很多基于微处理器中或微控制器的产品中广泛使用的通用型EPROM。基于8751的系统，其优点在于，可以把端口0和端口2作为输入输出端口用，而不是作为地址和数据总线来使用。但8751（30美元/片）比2764（5美元/片）的价格相对要昂贵些。

10.3.5　工厂掩模工艺

如果产品定型后，要进行大规模的生产，那么，相对于EPROM，诸如8051一类的工厂掩模ROM在成本上更划算。8051在功能上和8751等同。但有一点区别是，一旦固化完成后则再无法改写8051程序存储器中的代码。8051中的代码是在IC制造

过程中用掩模烧进（在IC制造的光刻阶段，基于版图做成的光学底片决定了激光的通过和阻塞）。光刻之后，8051中的各个存储单元或者处于激光通过状态，或者被阻塞，相应的各个ROM单元被赋值为1或0。

选用8751还是选用8051是一个经济成本的问题，工厂掩模ROM比EPROM便宜许多，但在制造工厂掩模ROM前，需要制作掩模以及启动掩模的生产流程，这笔前期费用为数不小。具体生产中能找到一个平衡点，根据该平衡点，可确定哪一种更可行。例如，假设8751的售价是25美元/片，而8051的售价是5美元/片，但还需要5000美元的前期费用。那么两种方法的平衡点可以按照如下式子计算出来：

```
25 n =5n+5000
20 n =5000
   n =250 units
```

如果8051的生产数超过250片，使用8051就比使用8751要划算。

如果要在8051和8031＋2764之间选择，问题就更复杂了。8031＋2764方案比起8751要便宜许多，因而，平衡点对应的数量就要大得多。假设8031＋2764的售价是7美元，平衡点就是：

```
7 n =5n+5000
2 n =5000
  n =2500 units
```

这样，若是只生产1000片8051，用8051就不划算了。但8031＋2764中使用了外部EPROM，这就意味着不能把端口0和端口2用作通用输入/输出端口了。在某些场合，这点很关键，可能会使开发者只好放弃8031＋2764方案。即使在输入/输出端口上没问题，开发者还要考虑其他因素。相比8051，8031＋2764方案要用到两片IC，这使得产品的制造、测试、维护、可靠性、芯片采购都变得复杂起来。还有许多其他因素，表面上看起来没什么关系，但却会实实在在地影响到8031＋2764产品的设计。而且还有一点要指出，基于8031＋2764方案的产品在尺寸上要大于8051产品。倘若要求最终产品要小尺寸，不管额外费用是多少，开发者都会选用8051来设计产品。

10.4　命令和开发环境

本节讨论开发环境在整体上是如何操作的。先说明1个观点，开发者在任何时候都是在某个“环境”下输入命令来完成工作。中心环境就是主机上的操作系统，绝大部分情况下，操作系统都是运行在PC机上的MS-DOS。如图10-4所示，在MS-DOS下输入命令，当某些命令执行完后，会返回MS-DOS，但另外一些命令执行后，会启动一个新的“环境”。

执行命令。命令分为两种，一种是**常驻内存命令**（如DIR等），一种是**非常驻命令**（如FORMAT、DISKCOPY等）。常驻命令在任何时候都处于内存中，准备着随

255 时被执行；非常驻命令是一个可执行的磁盘文件，若要执行该命令，需要先把文件读到内存中。

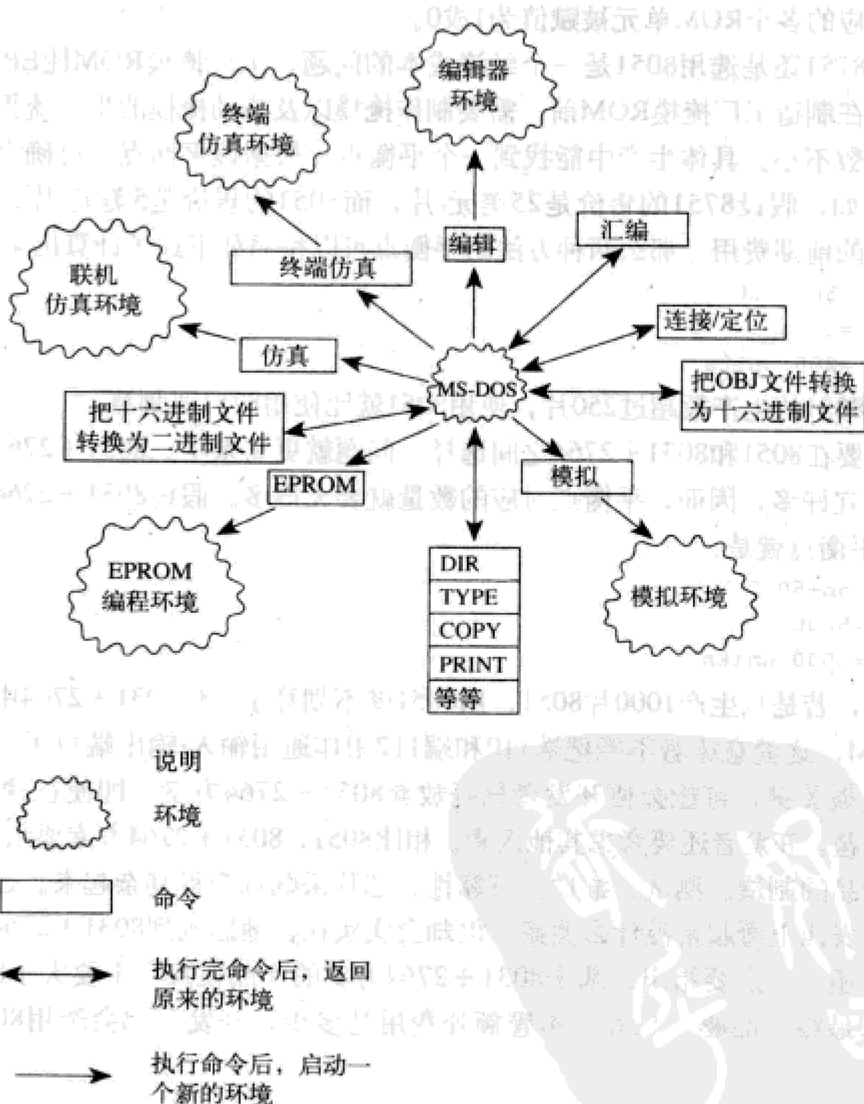


图10-4　开发环境

应用程序类似于非常驻命令，是磁盘上的一个可执行文件，要执行该程序，可以在MS-DOS提示符下直接键入文件名启动该程序。但不一定非要在MS-DOS提示符下才能调用该程序。命令或应用程序都可以在批处理文件中被调用，此外，二者还可以在菜单形式的用户界面下被调用，作为MS-DOS的前端程序而运行。

有多种方式可以输入命令执行所需要的参数。在典型情况下，参数紧跟在命令后面被输入，但也有其他输入方式，例如，某些命令带有各个参数的默认值，或者启动之后，程序提示用户输入参数等。但是，在给应用程序输入信息方面MS-DOS

中没有类似麦金（Macintosh）图形用户界面中的"对话框"那样的标准机制。

诸如"编辑器"这类的应用程序，在启动之后会"接管"系统，将用户带到一个新的环境中，以便和用户之间相交互。

环境。如图10-4所示，像仿真器、联机仿真器或者EPROM编程器这类软件工具，在调用之后，会启动自身所附带的环境。为了便于用户和环境交互，一般的环境都采用了快捷键、功能键、首字母大写的命令、高亮度显示的菜单项、默认路径等技术，但由于环境不同，所用的技术也有差别。用户若是要掌握这些细微差别，需要花费不少时间。当用户在环境里交互操作时，常常可以在各个环境之间切换。例如，用户可以从终端仿真器或编辑器里不时切换到DOS提示符下执行命令。执行完命令后，在MS-DOS提示符下输入EXIT命令，会立刻返回到刚才被挂起的环境里。

256

方法学。在人工智能和认知科学领域所做的研究发现，模仿人类解决问题的方式可以有助于取得更好的工作成果。人类在解决问题时，可以采用并行的方式认识事情的各个方面，并同时计算各种可能的行动，凭直觉进行处理。本章介绍的方法符合人类思维的本性。开发人员要对开发周期的各个流程以及开发环境所提供的各种工具和技术有全面的掌握，但要谨记，总体流程应当给予开发人员实质性的自主行事能力。

转换、查看、调用（启动一个新的环境）是开发过程的3种基本操作。对转换操作的结果要进行查看验证。开发人员对任何转换操作的结果（汇编、EPROM编程等）都要报以怀疑的态度，仔细查看结果，不要忽略任何细节。有很多工具可以用来校验转换后的结果，比如DIR命令（查看文件是否如预期般生成）、TYPE命令（查看生成的文件的内容）、EDIT命令、PRINT命令等。

小结

本章介绍了开发基于微控制器的产品时所用到的各种工具和技术。但书本知识无法代替实践经验。要把产品设计得成功，需要相当好的直觉。一个好的产品不是仅靠书本教出来的。开发者要把概念变成产品，最重要的就是要遵循一句话：试验并改正。

习题

10.1　如果8751 EPROM的单位售价是30美元，掩模编程的8051的单位售价是3美元外加10 000美元的前期启动费用。需求量要多大，才能使选用8051比选用8751更经济？如果一个项目需要生产3000个最终产品，选用8051要比采用8751节省多少美元？

10.2　下面是一个Intel十六进制格式的8051程序：

```
:10080000758911F007E0575A88AD28FD28D80FEF2
:10081000C28C758C3C758AB0DE087E050FBF09025C
```

257

```
:100820007F00D28C32048322FB90FC0CFC7AFCAD5E
:0A083000FD0AFD5CFDA6FDC8FDC831
:00000001FF
```

a. 程序的起始地址是多少?

b. 程序的长度是多少?

c. 程序的最后地址是多少?

10.3 下面是Intel十六进制文件中的某一行，但校验和有错，不正确的校验和出现在了最后两个字符“00”。正确的校验和应该是多少？

```
:100800007589117C007F0575A8FFD28FD28D80FE00
```

10.4 下面是一个Intel十六进制格式表示的8051程序：

```
:090100007820765508B880FA2237
:00000001FF
```

258

把这个程序对应的8051汇编代码写出来。

第11章 设计和接口实例

11.1 引言

本章通过几个项目设计和接口实例，进一步介绍8051微控制器软件和硬件方面的特点。第1个例子是基于8051的单板计算机——SBC-51的设计，它适合初学者学习8051或开发基于8051的产品。SBC-51里固化了一个监控程序，提供了一些系统操作的基本命令及用户指令。附录G列出了监控程序（MON51）的细节。

本章介绍的8051接口实例比前面各章里的例子要先进。每个例子都包含了硬件原理图、设计目标陈述、实现设计目标的程序代码清单以及硬件和软件操作的一般描述。程序代码的注释很详细，可通过注释了解软件的各种细节。

11.2 SBC-51

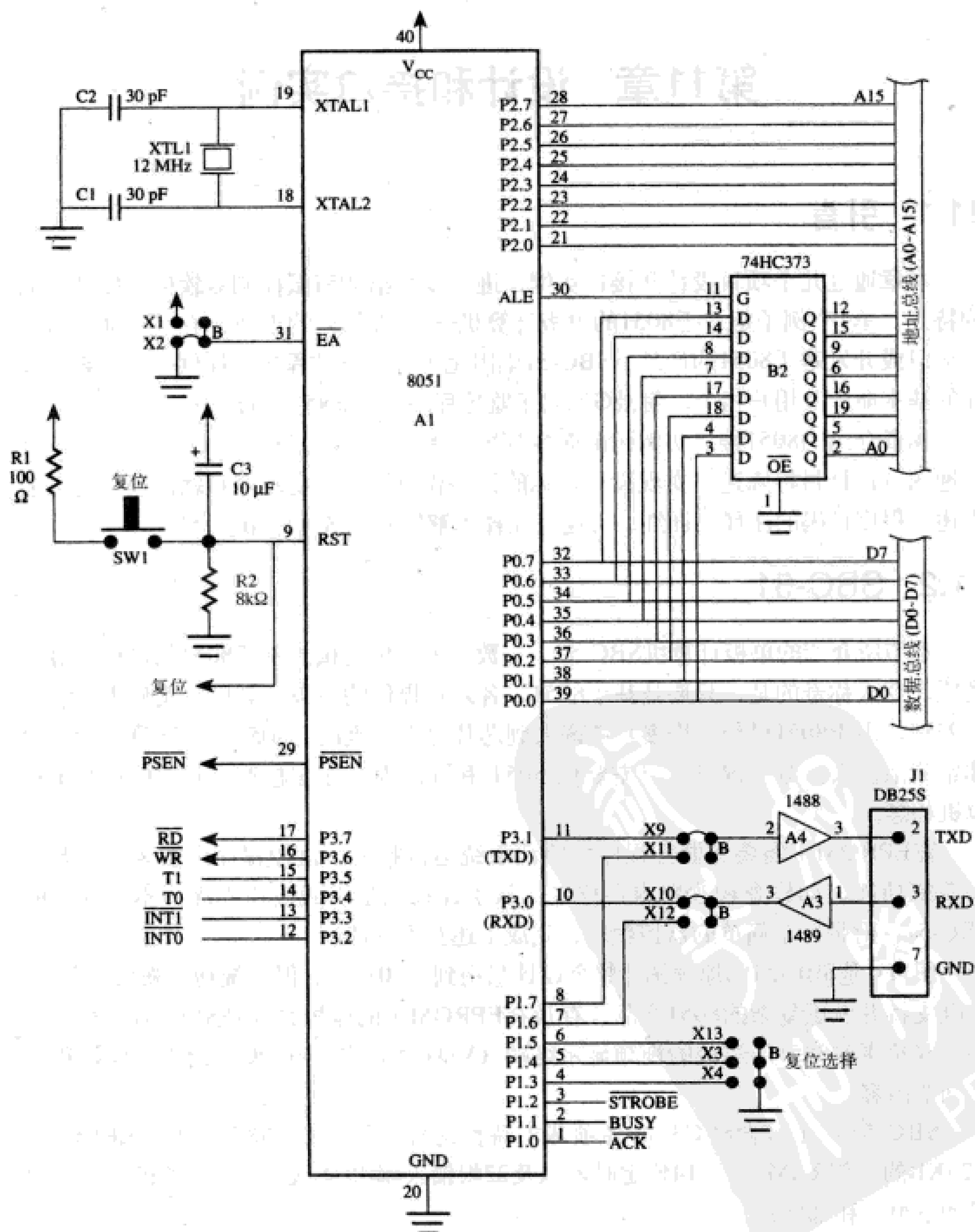
本节所介绍的单板计算机SBC-51，与数家公司所提供的基于8051的单板计算机类似。令人称奇的是，只要是基于8051，各公司提供的单板计算机在实质上没有什么差别。由于8051已经把许多功能嵌入到芯片之中，因而，8051单板计算机的设计非常简洁。大部分情况下，只需要把 8051 和外部内存连接起来，并将接口和主计算机相连。

在EPROM中需要有监控程序。要让系统运行起来，监控程序中需要有最基本的系统功能，如检查和改变内存位置、从主计算机上下载应用程序。本章介绍的SBC-51，附带一个简单的监控程序，完成上述基本功能。

图11-1是SBC-51的原理图，整个设计只用到了10片IC，但系统功能强大、灵活，足以支持开发更复杂的8051产品。存放在EPROM中的监控程序是SBC-51运行的核心，并负责与8051所连接的视频显示终端（VDT）通信。附录G中给出了监控程序的细节内容。

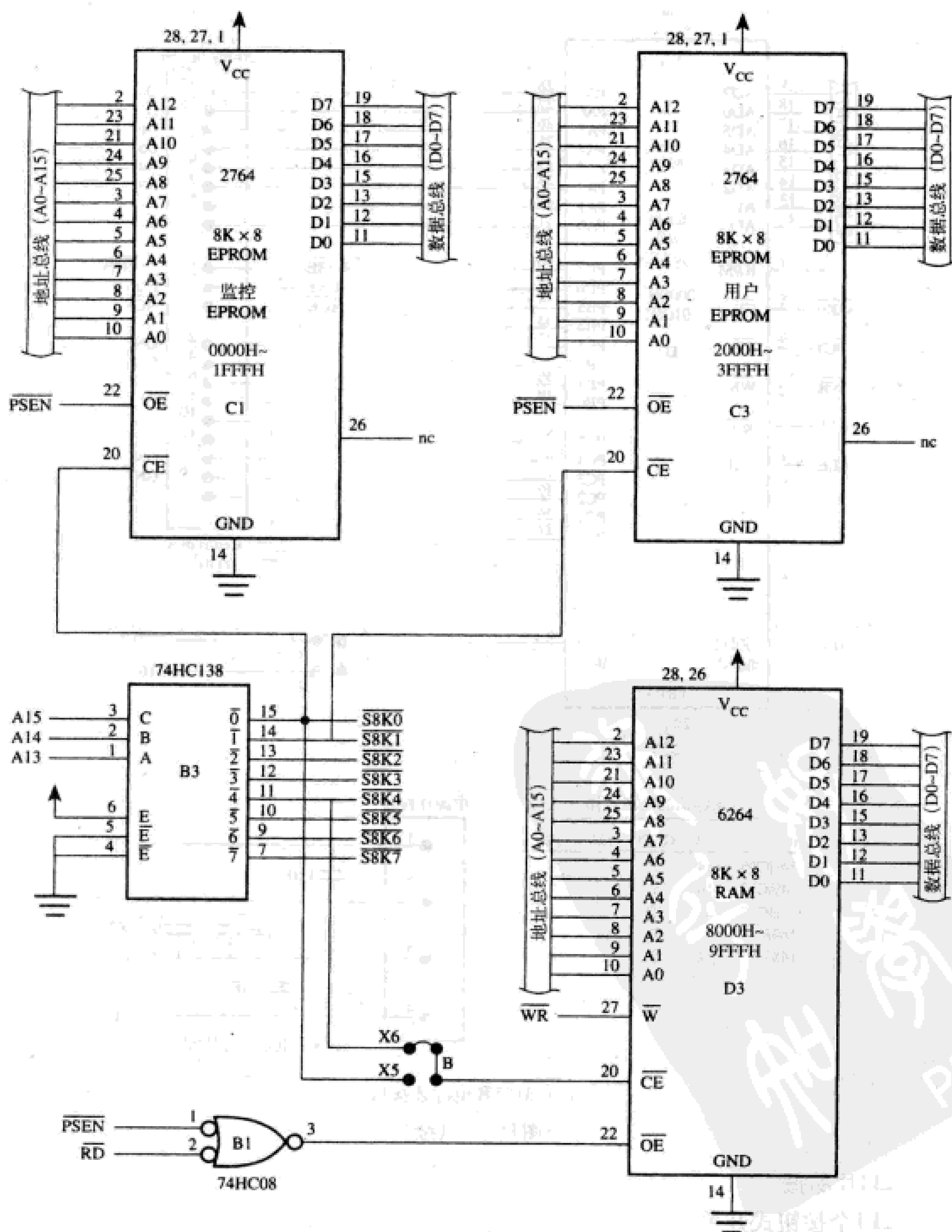
SBC-51除了包含80C31的各项基本特性之外，还包括16KB的外部EPROM、8.25KB的外部RAM、1个14位定时器以及22根输入/输出引线。图11-1的硬件配置包括如下组件和部件：

- ❑ 10片集成电路
- ❑ 15个电容
- ❑ 2个电阻



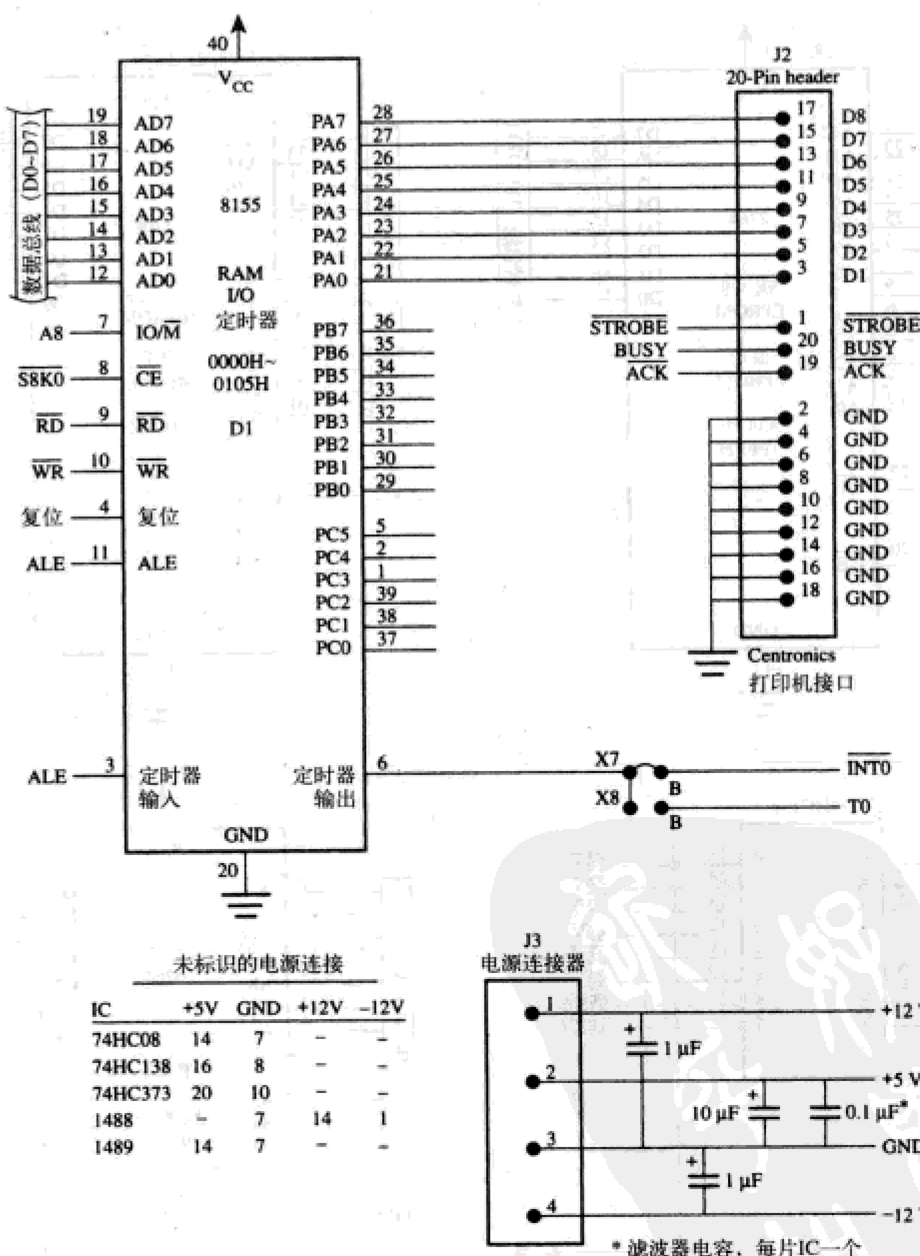
(a) 处理器和串行端口界面

图11-1 SBC-51 8051单板计算机——SBC-51



(b) 地址译码、RAM和EPROM

图11-1 （续）



未标识的电源连接

IC	+5V	GND	+12V	−12V
74HC08	14	7	–	–
74HC138	16	8	–	–
74HC373	20	10	–	–
1488	–	7	14	1
1489	14	7	–	–

（c）8155和电源连接口

图11-1 （续）

- ❑ 1片晶振
- ❑ 1个按钮式开关
- ❑ 3个连接器
- ❑ 13个配置跳线

设计中用到了外部存储器，因而端口0和端口2都不能用作输入/输出端口。虽然端口1和端口3的部分引脚有特殊用途，但根据系统的配置，端口1和端口3的其他引脚可用作输入/输出端口。

80C31通常用12MHz的晶振作为时钟源（请参考图2-2）。复位信号线（RST）在上电的时候由一个RC电路驱动进行复位，并且可用一个按钮开关来手动复位。端口0同时用作数据总线（D0~D7）和地址总线的低8位（A0~A7），如前文所述（参见2.7节）。在存取内存的时候，ALE信号驱动8位锁存器74HC373，锁存地址总线的低8位。80C31内部不包含片上ROM，从外部EPROM中执行程序，因而通过配置跳线X2将引脚 $\overline{EA}$ （外部访问启用）接地。

8051通过RS232C串行接口和主计算机或视频显示终端（VDT）相连接。图中的DB25S连接器按照数据终端设备（data terminal equipment，DTE）的规格连线，2号引脚传输信号线TXD，3号引脚接收信号线RXD，7号引脚接地。TXD上接有型号为1488的RS232驱动器，RXD上接有型号为1489的RS232驱动器。默认情况下，TXD和RXD通过跳线X9、X10和80C31相连接，TXD连到P3.1，RXD连到P3.0。但TXD和RXD还可以通过跳线 X11、X12分别和80C31的P1.7和P1.6相连接。

监控程序在复位的时候读取端口1上的引脚3、4和5的状态，监控程序会根据这3个引脚的状态决定是否启动特殊功能。但在系统复位后，这3个引脚可作为通用I/O端口使用。如果系统中用到了打印机，端口1上的引脚0、1和2用作打印机的握手信号；如果没有用到打印机，这3个引脚可作为通用I/O端口使用。

74HC138芯片对地址总线的3个高位地址端信号（A15~A13）进行译码，生成8个片选信号线（从/S8K0到/S8K7，每个片选信号线对应着8KB的存储器。但在该设计中，只用了4片IC：2片型号为2764的EPROM，1片型号为6264的RAM以及1片型号为8155的RAM/IO/TIMER。

260
~
263

图11-1中2片型号为2764的EPROM，单片容量是8KB。第1片2764（图中标示为"监控EPROM"）的片选信号线是 $\overline{S8K0}$ ，位于外部代码空间中地址0000H~lFFFH处。由于SBC-51在系统复位后，从地址0000H处立即开始执行程序，因而监控程序必须存放在该片2764中。第2片2764（图中标示为"用户EPROM"）的片选信号是 $\overline{S8K1}$ ，地址为2000H~3FFFH，用于存放用户的应用程序，对系统而言，这片EPROM不是必需的。请注意图中的2片2764的信号连线，如果要用， $\overline{CE}$ （片选使能信号，引脚20）和 $\overline{OE}$ 都必须有效（低电平）， $\overline{OE}$ 由80C31的 $\overline{PSEN}$ 信号驱动，因而，这2片2764位于8051的外部代码空间中。

芯片6264是容量为8KB的RAM，片选信号是 $\overline{S8K4}$ （如果跳线X6接上），因而6264对应的地址空间是8000H~9FFFH。用前面介绍的方法（请参考2.7.4节），可以让6264同时处于外部代码空间和外部数据空间中。因此，可以把用户程序加载（或写进）RAM中（认为RAM处于数据空间内），也可从RAM中执行用户程序（认为

RAM处于代码空间内)。

8155 RAM/IO/TIMER是一个外设接口芯片，用于验证SBC-51系统的扩展能力。采用类似的方法，还可以添加其他的外设接口芯片。8155对应的片选信号是 $\overline{S8K0}$，因而，8155位于存储器空间的低地址端。由于8155的“读”和“写”操作要使用 $\overline{RD}$ 和 $\overline{WR}$ 信号，因此，8155不会和存放监控程序的监控EPROM发生冲突（监控程序也是位于存储器空间的低地址端，但是在外部代码空间中）。

8155包含如下模块：

- ❑ 256B的RAM
- ❑ 22根输入/输出线
- ❑ 14位的定时器

8051的地址线A8连接到8155的IO/$\overline{M}$ 信号线（引脚7）上，当A8为低电平时，8155中的RAM功能被激活；当A8为高电平时，8155中的I/O信号线和定时器被激活。I/O信号线和定时器总计占据了6个不同的地址，因而，8155的地址从0000H~0105H (256+6)。分布如下：

地址	用途
0000H	第一个RAM字节的地址
…	其他RAM字节的地址
00FFH	最后一个RAM字节的地址
0100H	Interval/command寄存器
0101H	端口A
0102H	端口B
0103H	端口C
0104H	定时器中计时器的低8位
0105H	定时器中计数器的高6位+设定定时器模式的2位

264

生产厂家提供的产品规格清单中包含了8155芯片的功能细节，如果只是配置8155中的I/O端口，是非常容易的。系统复位后，默认情况下8155的所有端口信号线都是输入状态，因而，无需任何额外的初始化操作，就可以读取那些连到8155上的输入器件。例如，如果想把端口A的数据读取到累加器中，可以用下面的指令：

```
MOV   DPTR,#0101H        ;DPTR points to 8155 Port A
MOVX  A,@DPTR            ;read Port A into Acc
```

若要把端口A和端口B设置为输出模式，必须把8155的命令寄存器的第0位和第1位设置为1。例如，下面的指令把端口B配置为输出端口，端口A和端口C配置为输入端口：

```
MOV   DPTR,#0100H        ;8155 command register
MOV   A,#00000010B       ;Port B = output
MOVX  @DPTR,A            ;initialize 8155
```

设置命令寄存器的第2位和第3位的值为1，可以把端口C配置为输出端口。如下指令把3个端口都配置为输出端口：

```
MOV   DPTR,#0100H       ;8155 command register
MOV   A,#00001111B      ;all ports = output
MOVX  @DPTR,A           ;initialize 8155
```

图11-1中，8155的端口A和1个20针的接口（图中标示为“Centronics打印机接口”）相连，但打印机在此仅仅是作为接口例子说明而已。监控程序MON51中包括一个名为PCHAR（打印字符）的子例程，可用来把字符直接输出到VDT上，但如果在键盘上输入CONTROL-Z字符，那么将字符输出到并行打印机（请参考附录G）。当然，端口A也可另做他用。

图11-1中还给出了硬件系统的电源连接方式。数字器件在电平翻转时，由于电感效应，会引起+5V电源电压的波动，因而，在电路中添加滤波电容对于防止电压波动特别重要。如果SBC-51的原型设计是在面包板上（例如，采用导线连接）实现的，那么滤波电容对于硬件系统来说更是必不可少。设计时，要在电源接入点添加一个10μF的电解电容，在每个IC芯片的插座旁添加若干0.01μF的陶瓷电容，各个电容接在5V电压引脚和地之间。

SBC-51系统的规模很小，价格也不是很昂贵，读者很容易就可以构建出一个硬件原型，并且在阅读完本章中的各个例子以及附录F中的监控程序后，就可轻松地掌握如何设计SBC-51系统。构建SBC-51系统最实用的办法就是在面包板上布线，但开发者也可以在印制电路板上组装并测试SBC-51系统（请参考图11-2）①。

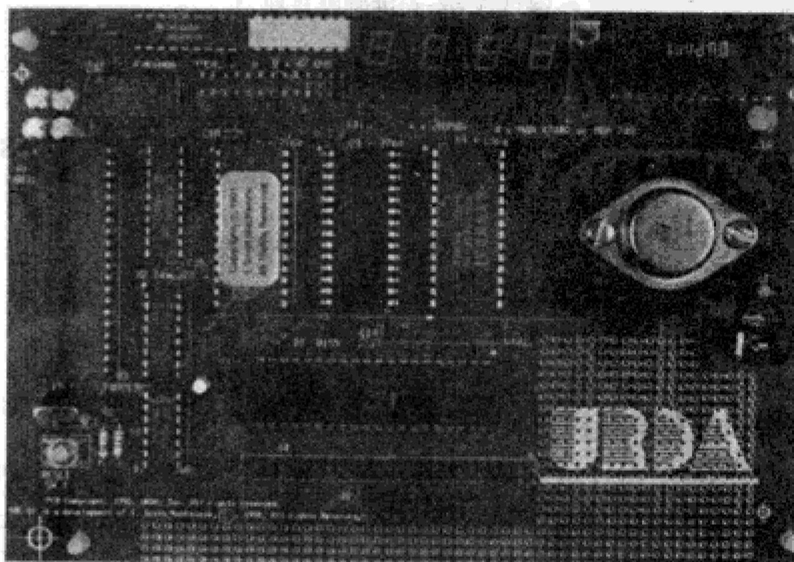


图11-2 SBC-51系统的印制电路板（得到URDA公司惠允）

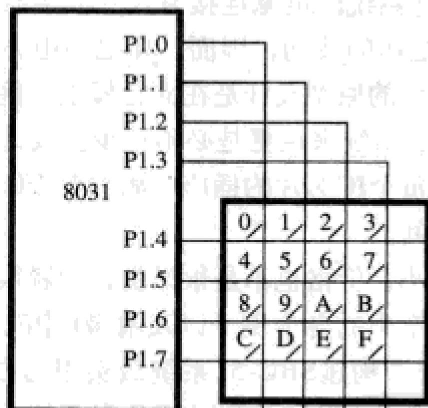
① SBC-51系统的印制电路板来自URDA公司，该公司位于美国匹兹堡（邮编15206）Jancey街道1811号200号房间。

SBC-51的介绍到此为止，下面各节讨论关于连接到SBC-51系统（或类似8051单板计算机系统）的外围器件接口的例子。

265

11.3　十六进制键盘接口

对于基于微控制器的设计，经常要用到键盘。用键盘和LED分别作为输入和输出设备，这种形式的用户接口是一种节俭的选择，通常对于较复杂的应用也可满足。例如微波炉和银行的自动取款机。图11-3给出了8031端口1和一个十六进制键盘的连线图。键盘上的十六个键按照四行四列的方式排布，行信号线接到端口1的第4~7位，列信号线连到端口1的第0~3位。



266

图11-3　十六进制键盘接口

例11-1　设计目标

写1个程序，连续地从键盘读取键值十六进制字符，并把相应的ASCII码显示到视频显示设备上（VDT）上。

这个例子看起来很简单，软件包括如下各个部分。

(1) 从键盘读取一个十六进制的键值字符。

(2) 把这个十六进制数转换为相应的ASCII码。

(3) 把ASCII码发送到VDT上。

(4) 回到第(1)步循环执行。

实际上，图11-4中程序清单的汇编代码（16~19行）就是按照上述流程组织的。当然，具体的功能都是以调用子例程的形式实现的。需要注意的是，上面的第(2)步和第(3)步是调用监控程序MON51中的子例程实现的。当然，可以从监控程序MON51中把对应的子例程代码复制出来，放到程序清单中，但这没有任何必要。程序清单前面定义了MON51两个子例程的入口地址（12~13行），新定义的符号是HTOA和OUTCHR，然后子例程在下面的主程序中可以按照常规方式被调用。附带

说一下，使用RL51程序把MON51模块链接和定位后，MON51子例程的入口地址将出现在RL5生成的符号表中。例如，在附录G中可以找到HTOA子例程和OUTCHR子例程的入口地址。

```
LOC   OBJ      LINE   SOURCE
                1     $DEBUG
                2     $NOPAGING
                3     $NOSYMBOLS
                4     ;FILE: KEYPAD.SRC
                5     ;*******************************************************
                6     ;               KEYPAD INTERFACE EXAMPLE              *
                7     ;                                                     *
                8     ; This program reads hexadecimal characters from a    *
                9     ; keypad attached to Port 1 and echos keys pressed    *
               10     ; to the console.                                     *
               11     ;*******************************************************
 033C          12     HTOA     EQU     033CH     ;MON51 subroutines (V12)
 01DE          13     OUTCHR   EQU     01DEH
               14
8000           15              ORG     8000H
8000 12800B    16     MAIN:    CALL    IN_HEX    ;get code from keypad
8003 12033C    17              CALL    HTOA      ;convert to ASCII
8006 1201DE    18              CALL    OUTCHR    ;echo to console
8009 80F5      19              SJMP    MAIN      ;repeat
               20
               21     ;*******************************************************
               22     ; IN_HEX - input hex code from keypad with debouncing *
               23     ;          for key press and key release (50 repeat   *
               24     ;          operations for each)                       *
               25     ;*******************************************************
800B 7B32      26     IN_HEX:  MOV     R3,#50    ;debounce count
800D 128022    27     BACK:    CALL    GET_KEY   ;key pressed?
8010 50F9      28              JNC     IN_HEX    ;no:  check again
8012 DBF9      29              DJNZ    R3,BACK   ;yes: repeat 50 times
8014 C0E0      30              PUSH    ACC       ;save hex code
8016 7B32      31     BACK2:   MOV     R3,#50    ;wait for key up
8018 128022    32     BACK3:   CALL    GET_KEY   ;key pressed?
801B 40F9      33              JC      BACK2     ;yes: keep checking
801D DBF9      34              DJNZ    R3,BACK3  ;no:  repeat 50 times
801F D0E0      35              POP     ACC       ;recover hex code and
8021 22        36              RET               ; return
               37
               38     ;*******************************************************
               39     ; GET_KEY - get keypad status                         *
               40     ;         - return with C = 0 if no key pressed       *
               41     ;         - return with C = 1 and hex code in ACC if  *
               42     ;           a key is pressed                          *
               43     ;*******************************************************
8022 74FE      44     GET_KEY: MOV     A,#0FEH           ;start with column 0
8024 7E04      45              MOV     R6,#4             ;use R6 as counter
8026 F590      46     TEST:    MOV     P1,A              ;activate colmn line
8028 FF        47              MOV     R7,A              ;save ACC
8029 E590      48              MOV     A,P1              ;read back Port 0
802B 54F0      49              ANL     A,#0F0H           ;isolate row lines
802D B4F007    50              CJNE    A,#0F0H,KEY_HIT   ;row line active?
8030 EF        51              MOV     A,R7              ;no: move to next
8031 23        52              RL      A                 ;    column line
8032 DEF2      53              DJNZ    R6,TEST
8034 C3        54              CLR     C                 ;no key pressed
```

图11-4 键盘接口软件

```
8035 8015    55              SJMP    EXIT         ;return with C = 0
8037 FF      56   KEY_HIT:   MOV     R7,A         ;save in R6
8038 7404    57              MOV     A,#4         ;prepare to caculate
803A C3      58              CLR     C            ; column weighting
803B 9E      59              SUBB    A,R6         ;4 - R6 = weighting
803C FE      60              MOV     R6,A         ;save in R6
803D EF      61              MOV     A,R7         ;restore scan code
803E C4      62              SWAP    A            ;put in low nibble
803F 7D04    63              MOV     R5,#4        ;use R5 as counter
8041 13      64   AGAIN:     RRC     A            ;rotate until 0
8042 5006    65              JNC     DONE         ;done when C = 0
8044 0E      66              INC     R6           ;add 4 until active
8045 0E      67              INC     R6           ; row found
8046 0E      68              INC     R6
8047 0E      69              INC     R6
8048 DDF7    70              DJNZ    R5,AGAIN
804A D3      71   DONE:      SETB    C            ;C = 1 (key pressed)
804B EE      72              MOV     A,R6         ;code in A (whew!!!)
804C 22      73   EXIT:      RET
             74              END
```

图11-4 (续)

在本例接口设计中，最具有难度的地方在于，如何编写子例程IN_HEX和GET_KEY的汇编代码？GET_KEY子例程负责扫描键盘的行和列，确定是否有键被按下。如果没有，该子例程设置C=0并返回；如果有，设置C=1，并把该键对应的十六进制代码读入到累加器A中，存放在第0~3位。

IN_HEX子例程的功能是软件去抖。键盘是由一系列的机械开关组合而成，开关在按下和松开的瞬间，接触点会产生多次抖动（开关接触点快速并短暂地重复闭合与断开）。去抖动原理如下：重复调用GET_KEY子例程，一直到调用的返回结果连续50次都是C=1，如果其中有1次调用的返回结果是C=0，那么认为当前仍然处于抖动过程中，程序复位计数器，重新从0开始计数。在确认某次按键有效后（连续50次C=1），IN_HEX继续调用GET_KEY子例程，直到调用的返回结果连续50次是C=0，这意味着按键已经被完全释放，程序可以再一次通过调用GET_KEY子例程来读入下一个按键结果了。

图11-4中的代码完成了前面所要求的功能，但实现的方法并不是太好。该段程序没有采用中断方式实现，若要把该段代码嵌入一个较大的应用程序，将会受到一些限制。因而，一种可行的改进方案是采用中断方式重新设计。下面的例子将会讨论一个中断驱动的接口设计。

267

11.4 多个七段LED的接口设计

第3章结束处介绍了一种七段LED显示器的接口设计（请参考图3-8）。但是，该设计占用了端口1上的7根信号线，就8051片上资源的分配策略而言，这种方法的效率很低。本节介绍一个包含4个七段LED接口的设计例子，该设计中仅使用了3根

8051的I/O信号线。很明显，本节中的设计有了非常大的改进，在那些需要使用多个LED的场合，采用改进后的方案也完全可以实现了。

设计的核心是一款摩托罗拉公司生产的MC14499芯片，是七段LED的译码器/驱动器，内部集成了驱动4个LED所需要的大部分电路。开发者仅需要额外添加如下部件：用于时序电路的0.015μF的电容，7个47 的限流电阻，4个型号为2N3904的三极管。图11-5中给出了80C51、MC14499以及4个七段LED的接线原理图。

例11-2 设计目标

假设在8051内部RAM中地址70H和71H存储单元中存有BCD数字，编写一段汇编程序实现，采用中断方式把这些数字输出到LED显示器上，要求每秒显示10次。

图11-6所示是完成上述任务的汇编代码。程序中用到了很多早先讨论过的概念。从较低层次看，子例程UPDATA和OUT8用于实现把数据输出到MC14499；从高层次看，该段代码阐明了如何设计一个中断驱动的应用程序，其中涉及很多的前台和后台操作（与第6章的例子不同，那里的程序仅在后台运行）。本例的中断程序和监控程序MON51共存于内存空间里，监控程序本身没有用到中断服务。监控程序在前台运行，而图11-6中的程序在中断发生的时候运行于后台。在启动执行图11-6中的程序后（在MON51控制台上输入GO 8000，请参考附录G），程序先是为LED显示器做些必需的中断初始化工作，然后立即把控制权交还给监控程序MON51。之后，在控制台上输入的监控程序命令照常执行，但此时后台不断产生中断。例如，倘若在监控程序控制台上用SET命令改变了内部RAM中地址70H和71H存储单元的内容，则在0.1s内，改变的内容就会立即显示在七段LED显示器上。

268 ~ 269

仔细分析程序清单中的整体结构，将发现下面各个部分依次出现：

- ❑ 汇编控制 (1~3行)
- ❑ 注释块 (4~30 行)
- ❑ 符号定义 (31~38 行)
- ❑ 定义存储空间 (40~42 行)
- ❑ 为程序和中断入口地址设立跳转表 (44~51 行)
- ❑ 主程序 (MAIN；56~69 行)
- ❑ 外部中断服务例程 (EXTOISR；74~77 行)
- ❑ 更新LED显示子例程 (UPDATE；89~97 行)
- ❑ 字节输出子例程 (OUT8；103~113行)
- ❑ 处理未实现的中断的代码 (118~123 行)

该程序将从SBC51的6264 RAM地址8000H开始执行。由于中断向量必须置于存储器的低端地址处，但监控程序在本应放置中断矢量的地址出放置了跳转表，把中断矢量重新定位到地址8000H开始的内存空间里（参考附录G）。程序的入口

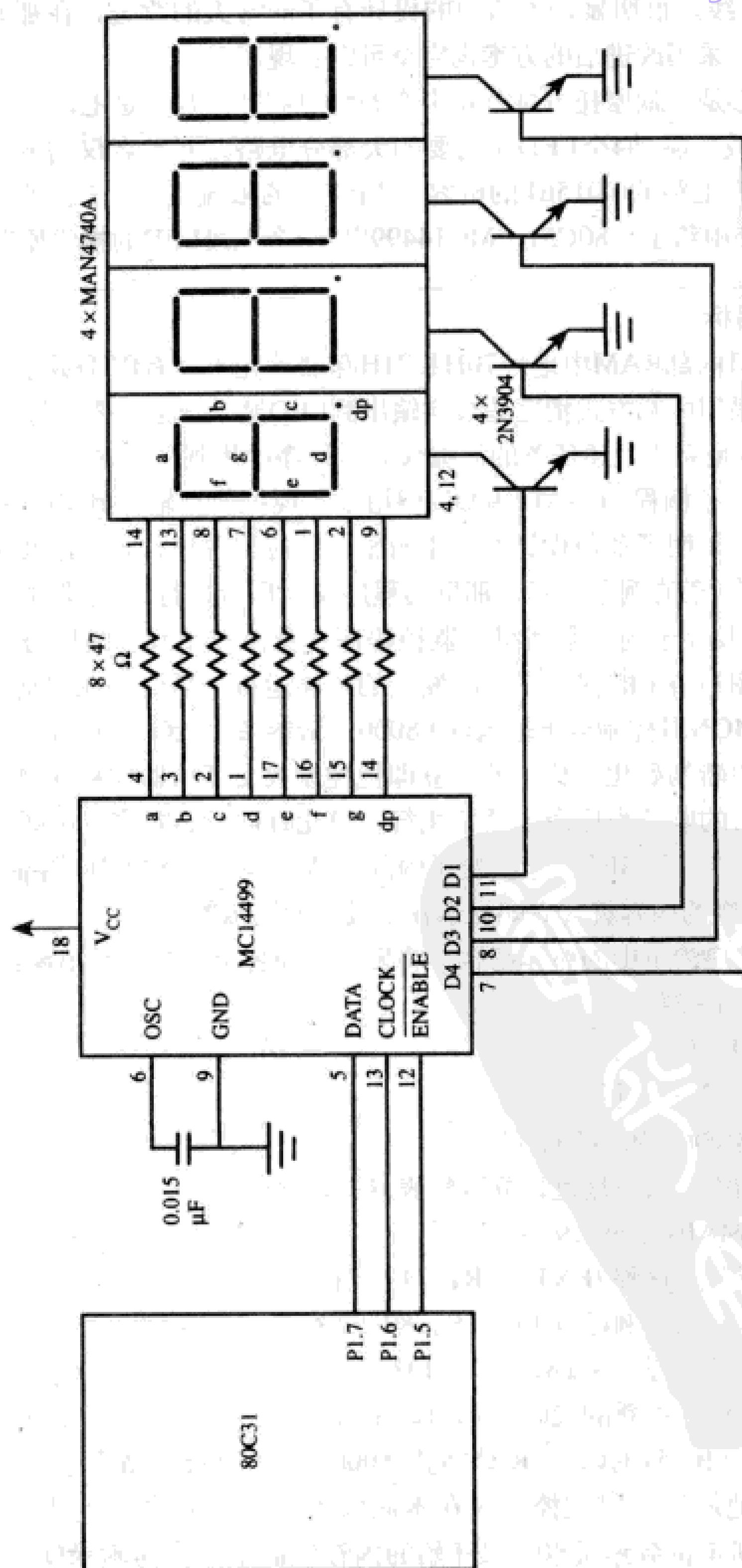


图11-5 MC14499和七段数码管的接线图

```
LOC   OBJ      LINE   SOURCE
                1     $DEBUG
                2     $NOPAGING
                3     $NOSYMBOLS
                4     ;FILE: MC14499.SRC
                5     ;*****************************************************************
                6     ;                  MC14499 INTERFACE EXAMPLE                    *
                7     ;                                                                *
                8     ; This program updates a 4-digit display 10 times per          *
                9     ; second using interrupts.  The digits are 7-segment           *
               10     ; LEDs driven by an MC14499 decoder/driver connected            *
               11     ; to P1.5 (-ENABLE), P1.6 (CLOCK), and P1.7 (DATA              *
               12     ; IN).  Interrupts are generated by the 8155's TIMER            *
               13     ; OUT line connected to -INT0.  TIMER OUT oscillates            *
               14     ; at 500 Hz and generates an interrupt on each 1-to-0          *
               15     ; transition.  An interrupt counter is used to update          *
               16     ; the display every 50 interrupts, for an update              *
               17     ; frequency of 10 Hz.                                           *
               18     ;                                                                *
               19     ; The example illustrates the foreground/background            *
               20     ; concept for interrupt-driven systems.  Once the              *
               21     ; 8155 is intitialized and External 0 interrupts are           *
               22     ; enabled, the program returns to the monitor program.*
               23     ; MON51 itself does not use interrupts; however, it            *
               24     ; executes as usual in the foreground while                    *
               25     ; interrupts take place in the background.  If the             *
               26     ; MON51 command SI (set internal memory) is used to            *
               27     ; change locations DIGITS or DIGITS+1, then the value          *
               28     ; written is immediately seen (within 0.1 s) on the            *
               29     ; LED display.                                                  *
               30     ;*****************************************************************
 00BC          31     MON51   CODE    00BCH        ;MON51 (V12) entry
 0100          32     X8155   XDATA   0100H        ;8155 address
 0104          33     TIMER   XDATA   X8155 + 4    ;timer registers
 0FA0          34     COUNT   EQU     4000         ;interrupts @ 2000 us
 0040          35     MODE    EQU     01000000B    ;timer mode bits
 0097          36     DIN     BIT     P1.7         ;MC14499 interface lines
 0096          37     CLOCK   BIT     P1.6
 0095          38     ENABLE  BIT     P1.5
               39
----           40             DSEG    AT 70H       ;absolute internal segment
0070           41     DIGITS: DS      2            ; (no conflict with MON51)
0072           42     ICOUNT: DS      1
               43
----           44             CSEG    AT 8000H
8000 028015    45             LJMP    MAIN         ; program entry point
8003 028031    46             LJMP    EX0ISR       ; 8155 interrupt
8006 02805D    47             LJMP    T0ISR        ; Timer 0 interrupt
8009 02805D    48             LJMP    EX1ISR       ; External 1 interrupt
800C 02805D    49             LJMP    T1ISR        ; Timer 1 interrupt
800F 02805D    50             LJMP    SPISR        ; Serial Port interrupt
8012 02805D    51             LJMP    T2ISR        ; Timer 2 interrupt
               52
               53     ;*****************************************************************
               54     ; MAIN PROGRAM BEGINS (INIT 8155 & ENABLE INTERRUPTS) *
               55     ;*****************************************************************
8015 900104    56     MAIN:   MOV     DPTR,#TIMER   ;initialize 8155 timer
8018 74A0      57             MOV     A,#LOW(COUNT)
801A F0        58             MOVX    @DPTR,A
801B A3        59             INC     DPTR          ;initialize high register
801C 744F      60             MOV     A,#HIGH(COUNT) OR MODE
```

图11-6 MC14499接口程序

```
801E F0          61              MOVX    @DPTR,A
801F 900100      62              MOV     DPTR,#X8155    ;8155 command register
8022 74C0        63              MOV     A,#0C0H        ;start timer command
8024 F0          64              MOVX    @DPTR,A        ;500 Hz square wave
8025 757232      65              MOV     ICOUNT,#50     ;initialize int. counter
8028 D2AF        66              SETB    EA             ;enable interrupts
802A D2A8        67              SETB    EX0            ;enable External 0 int.
802C D288        68              SETB    IT0            ;negative-edge triggered
802E 0200BC      69              LJMP    MON51          ;return to MON51
                 70
                 71      ;*********************************************************
                 72      ; EXTERNAL 0 INTERRUPT SERVICE ROUTINE                    *
                 73      ;*********************************************************
8031 D57205      74      EX0ISR: DJNZ    ICOUNT,EXIT    ;on 50th interrupt,
8034 757232      75              MOV     ICOUNT,#50     ; reset counter and
8037 113A        76              ACALL   UPDATE         ; refresh LED display
8039 32          77      EXIT:   RETI
                 78
                 79      ;*********************************************************
                 80      ; UDATE 4-DIGIT LED DISPLAY (EXECUTION TIME = 84 us)      *
                 81      ;                                                         *
                 82      ; ENTER:  Four BCD digits in internal memory              *
                 83      ;         locations DIGITS and DIGITS+1 (MSD in           *
                 84      ;         high nibble of DIGITS)                          *
                 85      ; EXIT:   MC14499 display updated                         *
                 86      ; USES:   P1.5, P1.6, P1.7                                *
                 87      ;         All memory locations and regs intact            *
                 88      ;*********************************************************
803A C0E0        89      UPDATE: PUSH    ACC          ;save Accumulator on stack
803C C295        90              CLR     ENABLE       ;prepare MC14499
803E E570        91              MOV     A,DIGITS     ;get first two digits
8040 114B        92              ACALL   OUT8         ;send two digits
8042 E571        93              MOV     A,DIGITS + 1 ;get second byte
8044 114B        94              ACALL   OUT8         ;send last two digits
8046 D295        95              SETB    ENABLE       ;disable MC14499
8048 D0E0        96              POP     ACC          ;restore ACC from stack
804A 22          97              RET
                 98
                 99      ;*********************************************************
                100      ; SEND 8 BITS IN ACCUMULATOR TO MC14499 (MSB FIRST)       *
                101      ;*********************************************************
                102              USING   0          ;assume reg. bank 0 enabled
804B C007       103      OUT8:   PUSH    AR7        ;save R7 on stack
804D 7F08       104              MOV     R7,#8      ;use R7 as bit counter
804F 33         105      AGAIN:  RLC     A          ;put bit in C flag
8050 9297       106              MOV     DIN,C      ;send it to MC14499
8052 C296       107              CLR     CLOCK      ;3 us low pulse on clock line
8054 00         108              NOP                ;NOPs needed to stretch pulse
8055 00         109              NOP                ; (minimum pulse width is
8056 D296       110              SETB    CLOCK      ; is 2 us)
8058 DFF5       111              DJNZ    R7,AGAIN   ;repeat until all 8 bits sent
805A D007       112              POP     AR7        ;restore R7 from stack
805C 22         113              RET
                114
                115      ;*********************************************************
                116      ; UNUSED INTERRUPTS (ERROR; RETURN TO MONITOR PROGRAM)    *
                117      ;*********************************************************
                118      T0ISR:
                119      EX1ISR:
                120      T1ISR:
                121      SPISR:
805D C2AF       122      T2ISR:  CLR     EA         ;shut off interrupts &
805F 0200BC     123              LJMP    MON51      ; return to MON51
                124              END
```

图11-6 (续)

地址一般设置为8000H，但是放置1条LJMP指令（参见图11-6的第45行），跳转到MAIN标号处开始执行。所有的初始化指令被安置在第56~68行。主程序以返回监控程序指令结束。

11.5 液晶显示（LCD）接口

前一个例子是关于七段LED显示的，只适于显示数字或简单的字符，但在一些需要显示比较复杂字符的场合，就需要采用其他类型的显示元件，比如液晶显示器。液晶显示器的一个较广泛的应用是科学计算器。在本节中，将给出一个液晶接口的例子，该液晶显示模块可以显示2行、每行16个字符，每个字符由5×7点阵构成。大部分LCD都是和事实上的行业标准日立公司的HD44780标准相兼容的。8051与HD44780兼容的LCD之间的连接都比较简单直观，如图11-7所示。

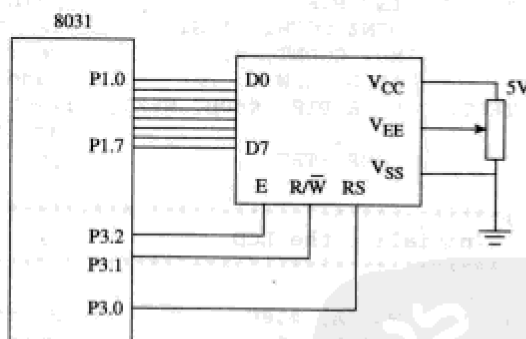


图11-7 LCD接口

270
~
273

例11-3 设计目标

假设ASCII字符存储在内部RAM的30H~7FH地址单元，编程实现，连续地在LCD上面显示这些ASCII字符，要求每次刷新16个字符。

该例子的程序清单如图11-8所示。LCD模块有3条控制线：寄存器选择（RS），读/写控制（R/$\overline{W}$）和启用（E）。当RS＝0时，表示将要发送到LCD模块的是命令；当RS＝1时，表示将要发送到LCD模块的是数据。当R/$\overline{W}$=1时表示8051将要从LCD读取数据；当R/$\overline{W}$=0时表示8051将要向LCD写数据。启用信号E需要设置为高电平，当其向低电平跳变时即通知LCD处理此时处于数据连线上的命令或字符数据。表11-1列出了LCD模块的所有命令码。

```
1        $DEBUG
2        $NOSYMBOLS
3        $NOPAGING
4        ;FILE: LCD.SRC
```

图11-8 LCD接口程序

```
                  5  ;*************************************************************
                  6  ;                    LCD INTERFACE EXAMPLE                  *
                  7  ;                                                           *
                  8  ; This program continually displays on the LCD              *
                  9  ; the ASCII characters stored in internal RAM               *
                 10  ; locations 30H-7FH.                                        *
                 11  ;*************************************************************
                 12
  00B0           13          RS      EQU             P3.0
  00B1           14          RW      EQU             P3.1
  00B2           15          E       EQU             P3.2
  0090           16          DBUS    EQU             P1
  REG            17          PTR     EQU             R0
  REG            18          COUNT   EQU             R1
                 19
8000             20          ORG 8000H
8000 1115        21  MAIN:   ACALL INIT              ;initialize LCD
8002 7916        22          MOV COUNT, #16          ;initialize char count
8004 7830        23  STRT:   MOV PTR, #30H           ;initialize pointer
8006 E6          24  NEXT:   MOV A, @PTR
8007 113A        25          ACALL DISP              ;display on LCD
8009 08          26          INC PTR                 ;point to next location
800A D904        27          DJNZ COUNT, TEST        ;is it end of line?
800C 7916        28          MOV COUNT, #16          ;yes: reinitialize count
800E 1127        29          ACALL NEW               ; and refresh LCD
8010 B880F3      30  TEST:   CJNE PTR, #80H, NEXT    ;last location?
                 31                                  ;no: go to next
8013 80EF        32          SJMP STRT               ;yes: go back to start
                 33
                 34  ;*************************************************************
                 35  ; Initialize the LCD                                        *
                 36  ;*************************************************************
                 37
8015 7438        37  INIT:   MOV A, #38H             ;2 lines, 5 x 7 matrix
8017 113D        38          ACALL WAIT              ;wait for LCD to be free
8019 C2B0        39          CLR RS                  ;output a command
801B 114B        40          ACALL OUT               ;send it out
                 41
801D 740E        42          MOV A, #0EH             ;LCD on, cursor on
801F 113D        43          ACALL WAIT              ;wait for LCD to be free
8021 C2B0        44          CLR RS                  ;output a command
8023 114B        45          ACALL OUT               ;send it out
                 46
8025 7401        47  NEW:    MOV A, #01H             ;clear LCD
8027 113D        48          ACALL WAIT              ;wait for LCD to be free
8029 C2B0        49          CLR RS                  ;output a command
802B 114B        50          ACALL OUT               ;send it out
                 51
802D 7480        52          MOV A, #80H             ;cursor: line 1, pos. 1
802F 113D        53          ACALL WAIT              ;wait for LCD to be free
8031 C2B0        54          CLR RS                  ;output a command
8033 114B        55          ACALL OUT               ;send it out
                 56
8035 22          57          RET
                 58
                 59  ;*************************************************************
                 60  ; Display data on LCD                                       *
                 61  ;*************************************************************
                 62
8036 113D        63  DISP:   ACALL WAIT              ;wait for LCD to be free
```

图11-8 （续）

```
8038 D2B0        64            SETB RS                  ;output a data
803A 114B        65            ACALL OUT                ;send it out
803C 22          66            RET
                 67
                 68     ;***********************************************************
                 69     ; Wait for LCD to be free                                  *
                 70     ;***********************************************************
803D C2B0        71     WAIT:  CLR RS                   ;command
803F D2B1        72            SETB RW                  ;read
8041 D297        73            SETB DBUS.7              ;DB7 = in
8043 D2B2        74            SETB E                   ;1-to-0 transition to
8045 C2B2        75            CLR E                    ; enable LCD
8047 2097F3      76            JB DBUS.7, WAIT
804A 22          77            RET
                 78
                 79     ;***********************************************************
                 80     ; Output to LCD                                            *
                 81     ;***********************************************************
804B F590        82     OUT:   MOV DBUS, A
804D C2B1        83            CLR RW
804F D2B2        84            SETB E
8051 C2B2        85            CLR E
8053 22          86            RET
                 87            END
```

图11-8 （续）

表11-1 LCD命令码

命令码	描述
1	清屏
2	返回初始位置
4	光标左移
5	显示右移
6	光标右移
7	显示左移
8	关显示，关光标
A	显示关，光标开
C	显示开，光标关
E	显示开，光标闪烁
F	显示开，光标不闪烁
10	移动光标至左侧
14	移动光标至右侧
18	将所有的显示内容左移
1C	将所有的显示内容右移
80	将光标强制移动到第1行的开始位置
C0	将光标强制移动到第2行的开始位置
38	2行，5×7点阵显示模式

该程序在依次显示1行16个ASCII字符之前，先调用INIT子例程初始化LCD显

274~276

示模块和设置。需要注意的是，每当向LCD发送数据之前，只要LCD处于忙碌状态，都必须先调用WAIT子例程。忙状态信息加载在LCD数据线的第7位。只有探测到LCD处于空闲状态时，程序才调用OUT子例程向LCD传送数据或命令。

11.6 扬声器接口

图11-9为8031和扬声器的接口。小型扬声器（如PC机或儿童玩具中的扬声器）由单个逻辑门即可驱动。扬声器线圈的一端接+5V，另一端连接到反相器74LS04的输出。电路中的反相器是必需的，它可以提高8031端口线的驱动能力。

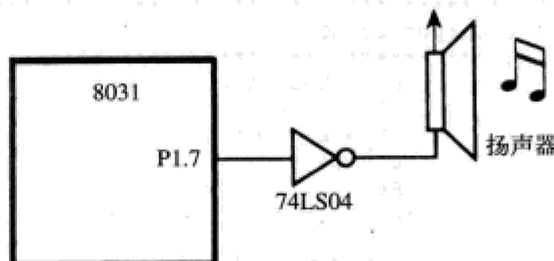


图11-9 扬声器和8031的接口

例11-4 设计目标

编写程序，要求采用中断方式驱动扬声器连续播放A大调。

8051很容易驱动扬声器产生各种音乐曲调。首先讨论一些相关的音乐理论，在图11-10程序清单的开始处（14~21行）的注释块中，列出了A大调所有音符所对应的频率。第1个音符的频率为440Hz，这是国际上为采用平均律校音的乐器（如钢琴）规定的参考频率。其他音符的频率都是该频率乘以$2^{n/12}$，其中n是各音符计算频率用的音级（音）。最简单的是A′，比A高八音，即12个音级，其频率为$440\times2^{12/12}=880$Hz。A′是本音级的最后一个音符。在任何大调音级中，各个音符和最低音符的音级数差依次是2、4、5、7、9、11和12。例如，程序清单中第18行的E音符，比最低音符高7个音级，故其频率为$440\times2^{7/12}=659.26$Hz。

为了产生一个音阶，需要两个时序：一个用于控制从一个音符到下一个音符的过渡；另一个控制驱动扬声器的端口线的翻转。这两个时序有很大的差别。例如要产生每秒4个音符的节奏，则每个音符的持续时间为250ms，即定时器溢出周期为250ms。如果在扬声器中要产生音阶中第一个音符的频率，则8051的输出端口必须在1.136ms（=1000/440/2）翻转1次（如图11-10第14行所示）。

程序的第43行将两个定时器都设置为16位模式，定时器0的中断负责改变音阶，定时器1的中断控制端口信号的翻转频率（即音阶的频率）。音阶频率相关的定时器重载值从90~104行的查找表中获得。更多的细节请参阅图11-10的程序清单。

277

```
                1    $debug
                2    $nopaging
                3    $nosymbols
                4    ;FILE: SCALE.SRC
                5    ;*********************************************************
                6    ;             LOUDSPEAKER INTERFACE EXAMPLE              *
                7    ;                                                        *
                8    ; This program plays an A major musical scale using     *
                9    ; a loudspeaker driven by a inverter through P1.7       *
               10    ;*********************************************************
               11    ;                                                        *
               12    ;  Note  Frequency (Hz)  Period (us)  Period/2 (us)     *
               13    ;  ----  --------------  -----------  -------------     *
               14    ;   A       440.00          2273          1136          *
               15    ;   B       493.88          2025          1012          *
               16    ;   C#      554.37          1804           902          *
               17    ;   D       587.33          1703           851          *
               18    ;   E       659.26          1517           758          *
               19    ;   F#      739.99          1351           676          *
               20    ;   G#      830.61          1204           602          *
               21    ;   A'      880.00          1136           568          *
               22    ;*********************************************************
 00BC          23    MONITOR  CODE   00BCH          ;MON51 (V12) entry point
 3CB0          24    COUNT    EQU    -50000         ;0.05 seconds per timeout
 0005          25    REPEAT   EQU    5              ;5 x 0.05 = 0.25 seconds/not
               26
               27    ;*********************************************************
               28    ; Note: X3 not installed on SBC-51, therefore           *
               29    ; interrupts directed to the following jump table       *
               30    ; beginning at 8000H                                     *
               31    ;*********************************************************
8000           32             ORG    8000H          ;RAM entry points for...
8000 028015    33             LJMP   MAIN           ; main program
8003 02806B    34             LJMP   EXT0ISR        ; External 0 interrupt
8006 028025    35             LJMP   T0ISR          ; Timer 0 interrupt
8009 02806B    36             LJMP   EXT1ISR        ; External 1 interrupt
800C 02803A    37             LJMP   T1ISR          ; Timer 1 interrupt
800F 02806B    38             LJMP   SPISR          ; Serial Port interrupt
8012 02806B    39             LJMP   T2ISR          ; Timer 2 interrupt
               40
               41    ;*********************************************************
               42    ; MAIN PROGRAM BEGINS                                    *
               43    ;*********************************************************
8015 758911    44    MAIN:    MOV    TMOD,#11H      ;both timers 16-bit mode
8018 7F00      45             MOV    R7,#0          ;use R7 as note counter
801A 7E05      46             MOV    R6,#REPEAT     ;use R6 as timeout counter
801C 75A88A    47             MOV    IE,#8AH        ;Timer 0 & 1 interrupts on
801F D28F      48             SETB   TF1            ;force Timer 1 interrupt
8021 D28D      49             SETB   TF0            ;force Timer 0 interrupt
8023 80FE      50             SJMP   $              ;ZzZzZzZz time for a nap
               51
               52    ;*********************************************************
               53    ; TIMER 0 INTERRUPT SERVICE ROUTINE (EVERY 0.05 SEC.)   *
               54    ;*********************************************************
8025 C28C      55    T0ISR:   CLR    TR0                      ;stop timer
8027 758C3C    56             MOV    TH0,#HIGH (COUNT)        ;reload
802A 758AB0    57             MOV    TL0,#LOW  (COUNT)
802D DE08      58             DJNZ   R6,EXIT                  ;if not 5th int, exit
802F 7E05      59             MOV    R6,#REPEAT               ;if 5th, reset
8031 0F        60             INC    R7                       ;increment note
8032 BF0C02    61             CJNE   R7,#LENGTH,EXIT          ;beyond last note?
8035 7F00      62             MOV    R7,#0                    ;yes: reset, A=440 Hz
8037 D28C      63    EXIT:    SETB   TR0                      ;no: start timer, go
8039 32        64             RETI                            ;    back to ZzZzZzZz
```

图11-10 扬声器接口程序

```
                  65
                  66        ;*****************************************************************
                  67        ; TIMER 1 INTERRUPT SERVICE ROUTINE (PITCH OF NOTES)            *
                  68        ;                                                               *
                  69        ; Note: The output frequencies are slightly off due             *
                  70        ; to the length of this ISR.  Timer reload values               *
                  71        ; need adjusting.                                               *
                  72        ;*****************************************************************
803A B297         73        T1ISR:  CPL   P1.7       ;music maestro!
803C C28E         74                CLR   TR1        ;stop timer
803E EF           75                MOV   A,R7       ;get note counter
803F 23           76                RL    A          ;multiply (2 bytes/note)
8040 128050       77                CALL  GETBYTE    ;get high-byte of count
8043 F58D         78                MOV   TH1,A      ;put in timer high register
8045 EF           79                MOV   A,R7       ;get note counter again
8046 23           80                RL    A          ;align on word boundary
8047 04           81                INC   A          ;past high-byte (whew!)
8048 128050       82                CALL  GETBYTE    ;get low-byte of count
804B F58B         83                MOV   TL1,A      ;put in timer low register
804D D28E         84                SETB  TR1        ;start timer
804F 32           85                RETI             ;time for a rest
                  86
                  87        ;*****************************************************************
                  88        ; GET A BYTE FROM LOOK-UP OF NOTES IN A MAJOR SCALE             *
                  89        ;*****************************************************************
8050 04           90        GETBYTE: INC  A          ;table look-up subroutine
8051 83           91                MOVC  A,@A+PC
8052 22           92                RET
8053 FB90         93        TABLE:  DW    -1136      ;A
8055 FB90         94                DW    -1136      ;A (play again; half note)
8057 FC0C         95                DW    -1012      ;B (quarter note, etc.)
8059 FC7A         96                DW    -902       ;C# - major third
805B FCAD         97                DW    -851       ;D
805D FD0A         98                DW    -758       ;E  - perfect fifth
805F FD5C         99                DW    -676       ;F#
8061 FDA6        100                DW    -602       ;G#
8063 FDC8        101                DW    -568       ;A'
8065 FDC8        102                DW    -568       ;A' (play 4 times; whole note)
8067 FDC8        103                DW    -568
8069 FDC8        104                DW    -568
  000C           105        LENGTH  EQU   ($ - TABLE) / 2 ;LENGTH = # of notes
                 106
                 107        ;*****************************************************************
                 108        ; UNUSED INTERRUPTS - BACK TO MONITOR PROGRAM (ERROR)           *
                 109        ;*****************************************************************
                 110        EXT0ISR:
                 111        EXT1ISR:
                 112        SPISR:
806B C2AF        113        T2ISR:  CLR   EA         ;shut off interrupts and
806D 0200BC      114                LJMP  MONITOR    ; return to MON51
                 115                END
```

图11-10 (续)

11.7 非易失性RAM接口

非易失性RAM（Nonvolatile RAM，NVRAM）是一种半导体存储器，其特点是存储的数据在掉电后不会丢失。NVRAM是标准静态RAM和可电擦除编程ROM

(EEPROM）的混合体。静态RAM中每1位数据都和EEPROM中的1位相重叠，数据可在二者之间传递。

NVRAM在基于微处理器和基于微控制器的应用中起到重要的作用。允许用户将那些偶尔需要改动，但又希望在掉电后仍然存在的数据或参数存储到NVRAM中。

在一些VDT的设计中，为了避免使用DIP式芯片插座容易造成数据无效的风险，转而选用NVRAM来存储像波特率、奇偶校验开/关、奇/偶校验等安装信息。当每次打开VDT时，将这些参数从NVRAM读到系统中，依次完成VDT的初始化。如果用户通过键盘改变了某些参数值，新参数会被保存到NVRAM内。

有自动拨号功能的调制解调器通常在其内存中存储若干电话号码。这些电话号码通常被保存在NVRAM中，所以即使突然掉电，电话号码仍然不会丢失。假设每个电话号码有7位数，如果采用BCD码，那么需要35个字节即可存储10个电话号码。

在本例中使用的NVRAM是Xicor①公司的X2444，Xicor是一家专门生产NVRAM和EEPROM器件的公司。X2444包含256bit静态RAM和与之相重叠的256bit EEPROM。数据可以在两个存储器之间进行传递。有两种传递方式供选择：微控制器通过串行接口发送相关指令，翻转引脚 $\overline{STORE}$ 和 $\overline{RECALL}$ 的输入状态。EEPROM中存放的数据是需要永久保存的数据，而RAM中的数据独立于EEPROM，是需要常常访问和更新的数据。关于X2444芯片的技术细节请参阅其数据表，如图11-11所示。

在本节的接口设计例子中，没有用到 $\overline{STORE}$ 和 $\overline{RECALL}$ 两个引脚。通过8051的端口引脚给X2444发送串行指令来设置其操作模式。

X2444和8051的接口如图11-12所示，仅用到了3根信号线：

- ❑ P1.0—SK（串行时钟）
- ❑ P1.1—CE（片选）
- ❑ P1.2—DI/DO（数据输入/输出）

当需要将指令传送到X2444时，先将CE置高，然后按时序将8位操作码通过DI/DO线发送到X2444。在本例中要用到如下操作码：

指令	操作码	操作
RCL	85H	把EEPROM中的数据转存到RAM
WREN	84H	设置写使能锁存
STORE	81H	把RAM数据转存到EEPROM
WRITE	1AAAA011B	写数据到RAM的AAAA地址
READ	1AAAA111B	从RAM的AAAA地址读数据

① XICOR, Inc.,851 Buckeye Count，Milpitas，CA 95035。

278 ~ 281



256位	商业应用	X2444 X24441	16×16位

非易失性静态RAM

特 征

- 单芯片微机的完美选择
  - 静态计时
  - 最少I/O接口
  - 串行端口兼容（COPS™，8051）
  - 易于微控制器端口连接
  - 最少的支持电路
- 非易失功能的软件和硬件控制
  - 最大限度的保存保护
- TTL兼容
- 16×16结构
- 低功耗
  - 有效电流：一般为15mA
  - 存储电流：一般为8mA
  - 旁路电流：一般为6mA
  - 静电流：一般为5mA
- 8引脚超小DIP封装

描 述

Xicor 2444是一个串行256位NOVRAM,其特征是带有一个16×16位的静态RAM，被一个非易失性E^2PROM矩阵位对位地覆盖。X2444是由具有同一可靠性的N沟道浮动栅MOS工艺制造的；所有Xicor 5V非易失性内存都是使用该项工艺制作的

Xicor NOVRAM设计允许通过软件命令或外部硬件输入在2个内存阵列间传送数据。可以在10ms甚至更短时间内完成存储操作（RAM数据存入E^2PROM）；重新调用数据操作则可以在2.5μs甚至更短时间内完成

Xicor NOVRAM是为不受限地向RAM写数据操作而设计的，无论是从主机写数据还是从E^2PROM调用数据均可，至少也可以完成100 000次存储操作。数据可以保存至少100年

COPS™是National Semiconductor公司的商标

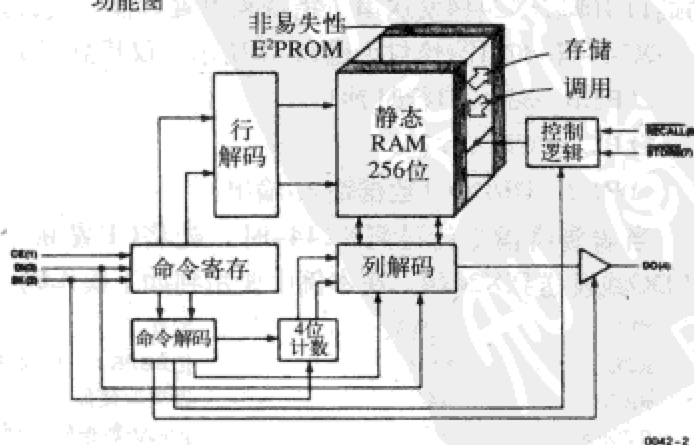
引脚配置



引脚名称

CE	芯片启用
SK	串行时钟
DI	串行数据输入
DO	串行数据输出
RECALL	调用
STORE	存储
V_{CC}	+5V
V_{SS}	地

功能图



May 1987

1

图11-11 X2444 非易失RAM芯片数据表

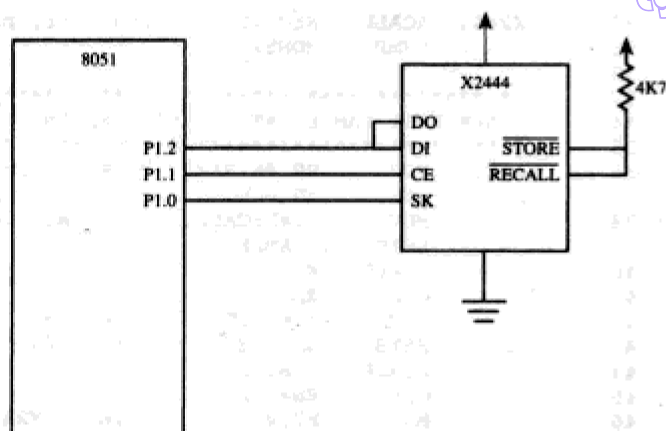


图11-12 X2444 非易失RAM的接口

例11-5 设计实例

编写2个程序，第1个名为SAVE，将8051内部RAM 60H~7FH单元的内容复制到X2444的EEPROM。第2个名为RECOVER，将先前存储在X2444 EEPROM中的内容读回，并存放在内部RAM的60H~7FH单元。

这两个程序分别应用在不同的场合下。SAVE程序一般用于非易失信息需要修改的时候（例如，在用户改变某配置参数时）。RECOVER程序在每次设备上电或复位时被执行。在本例中，非易失信息被保存在内部RAM的60H~7FH单元（假定这些信息是以固件形式存在的控制程序所需要的一些参数）。程序清单请参阅图11-13。

```
                1        $DEBUG
                2        $NOPAGING
                3        $NOSYMBOLS
                4        ;FILE: NVRAM.SRC
                5        ;*************************************************************
                6        ;                  X2444 INTERFACE EXAMPLE                  *
                7        ;                                                           *
                8        ; Two subroutines are shown below that SAVE or              *
                9        ; RECOVER data between a X2444 non-volatile RAM and         *
               10        ; 32 bytes of the 8051's internal RAM.                      *
               11        ;*************************************************************
  0085         12        RECALL  EQU     85H     ;X2444 recall instruction
  0084         13        WRITE   EQU     84H     ;X22444 write enable instruction
  0081         14        STORE   EQU     81H     ;X2444 store instruction
  0082         15        SLEEP   EQU     82H     ;X2444 sleep istruction
  0083         16        W_DATA  EQU     83H     ;X2444 write data instruction
  0087         17        R_DATA  EQU     87H     ;X2444 read data instruction
  00BC         18        MON51   EQU     00BCH   ;MON51 entry point (V12)
  0020         19        LENGTH  EQU     32      ;32 bytes saved/restored
  0092         20        DIN     BIT     P1.2    ;X2444 interface lines
  0091         21        ENABLE  BIT     P1.1
  0090         22        CLOCK   BIT     P1.0
               23
----           24                DSEG    AT 60H
0060           25        NVRAM:  DS      LENGTH    ;60H-7FH saved/recovered
               26
----           27                CSEG    AT 8000H
8000 110A      28        WX2444: ACALL   SAVE      ;8000H entry point for write
8002 0200BC    29                LJMP    MON51
```

图11-13 X2444接口程序清单

```
8005 1149      30  RX2444: ACALL   RECOVER    ;8005H entry point for read
8007 0200BC    31          LJMP    MON51
               32
               33  ;***********************************************************
               34  ; SAVE 8031 RAM LOCATIONS 60H-7FH IN X2444 NVRAM           *
               35  ;***********************************************************
800A 7860      36  SAVE:   MOV     R0,#NVRAM      ;R0 -> locations to save
800C C291      37          CLR     ENABLE         ;disable X2444
800E 7485      38          MOV     A,#RECALL      ;recall instruction
8010 D291      39          SETB    ENABLE
8012 1184      40          ACALL   W_BYTE
8014 C291      41          CLR     ENABLE
8016 7484      42          MOV     A,#WRITE       ;write enable prepares
8018 D291      43          SETB    ENABLE         ; X2444 to be written to
801A 1184      44          ACALL   W_BYTE
801C C291      45          CLR     ENABLE
801E 7F00      46          MOV     R7,#0          ;R7 = X2444 address
8020 EF        47  AGAIN:  MOV     A,R7           ;put address in ACC
8021 23        48          RL      A              ;put in bits 3,4,5,6
8022 23        49          RL      A
8023 23        50          RL      A
8024 4483      51          ORL     A,#W_DATA      ;build write instruction
8026 D291      52          SETB    ENABLE
8028 1184      53          ACALL   W_BYTE
802A 7D02      54          MOV     R5,#2
802C E6        55  LOOP:   MOV     A,@R0          ;get 8051 data
802D 08        56          INC     R0             ;point to next byte
802E 1184      57          ACALL   W_BYTE         ;send byte to X2444
8030 DDFA      58          DJNZ    R5,LOOP        ;repeat (send 2nd byte)
8032 C291      59          CLR     ENABLE
8034 0F        60          INC     R7             ;increment X2444 address
8035 BF10E8    61          CJNE    R7,#16,AGAIN   ;if not finished, again
8038 7481      62          MOV     A,#STORE       ;if finished, copy to EEPR
803A D291      63          SETB    ENABLE
803C 1184      64          ACALL   W_BYTE
803E C291      65          CLR     ENABLE
8040 7482      66          MOV     A,#SLEEP        ;put X2444 to sleep
8042 D291      67          SETB    ENABLE
8044 1184      68          ACALL   W_BYTE
8046 C291      69          CLR     ENABLE
8048 22        70          RET                     ;DONE!
               71
               72  ;***********************************************************
               73  ; RECOVER 8051 RAM LOCATIONS 60H-7FH FROM X2444 NVRAM *
               74  ;***********************************************************
8049 7860      75  RECOVER: MOV    R0,#NVRAM
804B C291      76          CLR     ENABLE
804D 7485      77          MOV     A,#RECALL       ;recall instruction
804F D291      78          SETB    ENABLE
8051 1184      79          ACALL   W_BYTE
8053 C291      80          CLR     ENABLE
8055 7F00      81          MOV     R7,#0           ;R7 = X2444 address
8057 EF        82  AGAIN2: MOV     A,R7            ;put address in ACC
8058 23        83          RL      A               ;build read instruction
8059 23        84          RL      A
805A 23        85          RL      A
805B 4487      86          ORL     A,#R_DATA
805D D291      87          SETB    ENABLE
805F 1184      88          ACALL   W_BYTE          ;send read instruction
8061 7D02      89          MOV     R5,#2           ; (+ address)
8063 1178      90  LOOP2:  ACALL   R_BYTE          ;read byte of data
8065 F6        91          MOV     @R0,A           ;put in 8051 RAM
8066 08        92          INC     R0              ;point to next location
```

图11-13 （续）

```
8067 DDFA      93             DJNZ   R5,LOOP2
8069 C291      94             CLR    ENABLE
806B 0F        95             INC    R7              ;increment X2444 address
806C BF10E8    96             CJNE   R7,#16,AGAIN2   ;repeat until last
806F 7482      97             MOV    A,#SLEEP        ;put X2444 to sleep
8071 D291      98             SETB   ENABLE
8073 1184      99             ACALL  W_BYTE
8075 C291     100             CLR    ENABLE
8077 22       101             RET                    ;DONE!
              102
              103    ;*********************************************************
              104    ; READ A BYTE OF DATA FROM X2444                          *
              105    ;*********************************************************
8078 7E08     106    R_BYTE:  MOV    R6,#8        ;use R6 as bit counter
807A A292     107    AGAIN3:  MOV    C,DIN        ;put X2444 data bit in C
807C 33       108             RLC    A            ;build byte in Accumulator
807D D290     109             SETB   CLOCK        ;toggle clock line (1 us)
807F C290     110             CLR    CLOCK
8081 DEF7     111             DJNZ   R6,AGAIN3    ;if not last bit, do again
8083 22       112             RET
              113
              114    ;*********************************************************
              115    ; WRITE A BYTE OF DATA TO X2444                           *
              116    ;*********************************************************
8084 7E08     117    W_BYTE:  MOV    R6,#8        ;use R6 as bit counter
8086 33       118    AGAIN4:  RLC    A            ;put bit to write in C
8087 9292     119             MOV    DIN,C        ;put in X2444 DATA IN line
8089 D290     120             SETB   CLOCK        ;clock bit into X2444
808B C290     121             CLR    CLOCK
808D DEF7     122             DJNZ   R6,AGAIN4    ;if not last bit, do again
808F 22       123             RET
              124             END
```

图11-13 （续）

程序中的存储操作和取回数据操作包括如下步骤：

写数据到X2444

(1) 执行RCL指令

(2) 执行WREN指令

(3) 写数据到X2444的RAM中

(4) 执行STO指令

(5) 执行SLEEP指令

从X2444读取数据

(1) 执行RCL指令

(2) 从X2444的RAM读取数据

(3) 执行SLEEP指令

为了说明程序代码是如何完成这些操作的，图11-14给出了向X2444发送RCL指令的时序图。有几个数据位是无关紧要的（具体可参考X2444的数据表），需要注意的是，它们在时序图中都显示为0。

写数据和读数据的时序稍微有些差别，如果紧随8位操作码之后的是16位的数

据，片选信号CE要保持高电平24位的时间。对于读指令，首先将8位操作码写到X2444，然后再从X2444读取数据。读8位数据（R_BYTE, 106~112行）和写8位数据（W_BYTE, 117~123行）是2个独立的子例程。具体细节请参阅程序清单。

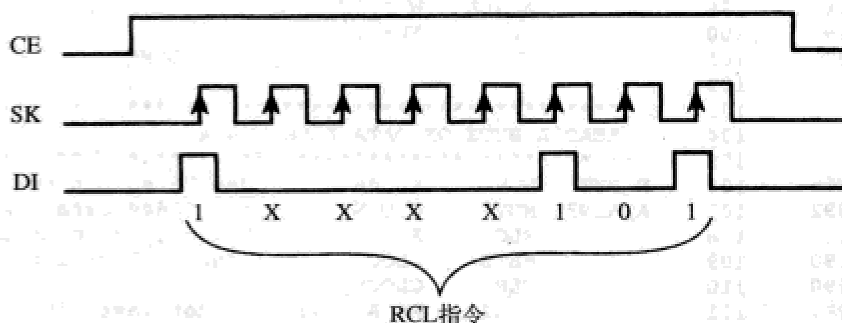


图11-14 X2444 RCL指令的时序

11.8 输入输出的扩展

8051具有端口0~端口3共4个输入输出端口。如果系统中扩展了外部存储器，那么端口0和端口2的全部或部分I/O口线需要作为地址/数据总线来使用，因此，用于通用I/O的口线减少。本节将讨论2种简单的I/O扩展方法。

282~284

11.8.1 使用移位寄存器

本例将介绍1个增加8051输入口线的简单方法。3条口线用于连接多片（在本例中为2片）并入串出移位寄存器74HC165（如图11-15所示）。将SHIFT/ $\overline{\text{LOAD}}$ 设置为低电平，就可以把外部输入信号并行读入到74HC165中。然后通过读取DATA IN线将数据读入8051，并给CLOCK线加上时钟信号。每个时钟脉冲将数据移位（向下，如图11-15所示），所以数据被1位接1位地读入到8051中。

例11-6 设计目标

编写子例程，将图11-15中16位输入状态复制到8051内部RAM的25H和26H单元。

完成该任务的程序清单如图11-16所示。在主程序循环中调用两个子例程：GET_BYTES和DISPLAY_RESULTS（34~35行）。后面的子例程介绍了一种在资源有限的情况下非常有用的调试技术。DISPLAY_RESULTS（72~83行）子例程从8051内部RAM的25H和26H单元读取数据，然后以十六进制字符的形式发送到控制台上。这为我们提供了一种简单的用于验证程序和接口是否正常工作的可视化技术途径。当输入线的状态在高、低电平间切换时，如果接口和程序都在正常工作的话，这种输入状态的改变将立即显示在控制台上。

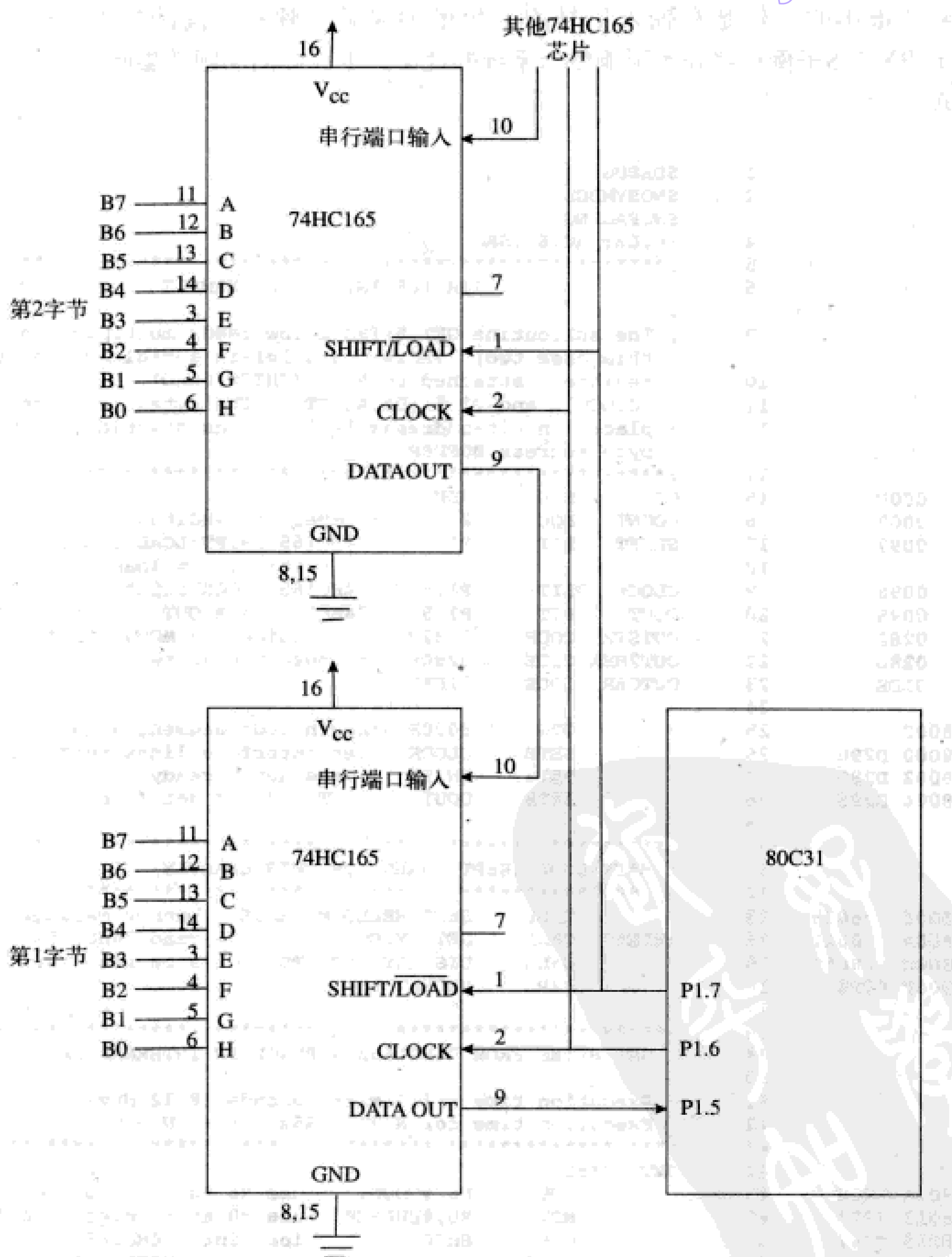


图11-15 2片74HC165接口设计

在使用两片74HC165、系统的晶振频率为12MHz的情况下，GET_BYTES子例程（44~58行）的执行时间为112μs。如果系统对输入信号进行每秒20次采样，GET_BYTES将消耗CPU 0.2%的执行时间（112/50000＝0.2%）。这对系统资源的

影响是很小的，但是在输入信号的个数增加或采样频率提高的情况下，执行GET_BYTES子例程将开始影响整个系统的性能。更详细的问题请参阅下面的程序清单。

```
                   1      $DEBUG
                   2      $NOSYMBOLS
                   3      $NOPAGING
                   4      ;FILE: HC165.SRC
                   5      ;********************************************************
                   6      ;                74HC165 INTERFACE EXAMPLE              *
                   7      ;                                                       *
                   8      ; The subroutine GET_BYTES below reads multiple (in     *
                   9      ; this case two) 74HC165 parallel-in serial-out shift   *
                  10      ; registers attached to P1.7 (SHIFT/LOAD), P1.6         *
                  11      ; (CLOCK), and P1.5 (DATA OUT).  The bytes read are     *
                  12      ; placed in bit-addressable locations starting at the   *
                  13      ; byte address BUFFER.                                  *
                  14      ;********************************************************
   000D           15      CR       EQU      0DH
   0002           16      COUNT    EQU      2         ;number of 74HC165s
   0097           17      SHIFT    BIT      P1.7      ;74HC165 SHIFT/LOAD input
                  18                                  ; 1 = shift, 0 = load
   0096           19      CLOCK    BIT      P1.6      ;74HC165 CLOCK input
   0095           20      DOUT     BIT      P1.5      ;74HC165 DATA OUT output
   0282           21      OUTSTR   CODE     0282H     ;subroutines in MON51(V12)
   028D           22      OUT2HEX  CODE     028DH     ;output byte as two hex char.
   01DE           23      OUTCHR   CODE     01DEH
                  24
 8000             25               ORG      8000H     ;begin code segment at 8000H
 8000 D296        26               SETB     CLOCK     ;set interface lines initially in
 8002 D297        27               SETB     SHIFT     ; case not already
 8004 D295        28               SETB     DOUT      ;DOUT must be set (input)
                  29
                  30      ;********************************************************
                  31      ; MAIN LOOP (KEPT SMALL FOR THIS EXAMPLE)               *
                  32      ;********************************************************
 8006 128029      33               CALL     SEND_HELLO_MESSAGE  ;banner message
 8009 128011      34      REPEAT:  CALL     GET_BYTES           ;read 74HC165s
 800C 128050      35               CALL     DISPLAY_RESULTS     ;show results
 800F 80F8        36               JMP      REPEAT              ;loop
                  37
                  38      ;********************************************************
                  39      ; GET BYTES FROM 74HC165s & PLACE IN INTERNAL RAM       *
                  40      ;                                                       *
                  41      ; Execution time = 112 microseconds (@ 12 MHz).         *
                  42      ; Execution time for N 74HC165s = 6 + (N x 53) us       *
                  43      ;********************************************************
                  44      GET_BYTES:
 8011 7E02        45               MOV      R6,#COUNT      ;use R6 as byte counter
 8013 7825        46               MOV      R0,#BUFFER     ;use R0 as pointer to buffer
 8015 C297        47               CLR      SHIFT          ;load into 74HC165s by
 8017 D297        48               SETB     SHIFT          ; pulsing SHIFT/LOAD low
 8019 7F08        49      AGAIN:   MOV      R7,#8          ;use R7 as bit counter
 801B A295        50      LOOP:    MOV      C,DOUT         ;get a bit (put it in C)
 801D 13          51               RRC      A              ;put in ACC.0 (LSB 1st)
 801E C296        52               CLR      CLOCK          ;pulse CLOCK line (shifts
 8020 D296        53               SETB     CLOCK          ; bits toward DATA OUT)
 8022 DFF7        54               DJNZ     R7,LOOP        ;if not 8th shift, repeat
 8024 F6          55               MOV      @R0,A          ;if 8th shift, put in buf.
```

图11-16 74HC165接口程序清单

```
8025 08          56              INC     R0              ;increment pointer to buf.
8026 DEF1        57              DJNZ    R6,AGAIN        ;get two bytes
8028 22          58              RET
                 59
                 60      ;*************************************************************
                 61      ; SEND HELLO MESSAGE TO CONSOLE (DEBUGGING AID)              *
                 62      ;*************************************************************
                 63      SEND_HELLO_MESSAGE:
8029 908030      64              MOV     DPTR,#BANNER    ;point to hello message
802C 120282      65              CALL    OUTSTR          ;send it to console
802F 22          66              RET
8030 2A2A2A20    67      BANNER: DB      '*** TEST 74HC165 INTERFACE ***',CR,0
8034 54455354
8038 20373448
803C 43313635
8040 20494E54
8044 45524641
8048 4345202A
804C 2A2A
804E 0D
804F 00
                 68
                 69      ;*************************************************************
                 70      ; DISPLAY RESULTS ON CONSOLE (DEBUGGING AID)                 *
                 71      ;*************************************************************
                 72      DISPLAY_RESULTS:                        ;display bytes
8050 7825        73              MOV     R0,#BUFFER      ;R0 points to bytes
8052 7E02        74              MOV     R6,#COUNT       ;R6 is # of bytes read
8054 E6          75      LOOP2:  MOV     A,@R0           ;get byte
8055 08          76              INC     R0              ;increment pointer
8056 12028D      77              CALL    OUT2HEX         ;output as 2 hex char.
8059 7420        78              MOV     A,#' '          ;separate bytes
805B 1201DE      79              CALL    OUTCHR
805E DEF4        80              DJNZ    R6,LOOP2        ;repeat for each byte
8060 740D        81              MOV     A,#CR           ;begin a new line
8062 1201DE      82              CALL    OUTCHR          ;send CR (LF too!)
8065 22          83              RET
                 84
                 85      ;*************************************************************
                 86      ; CREATE BUFFER IN BIT-ADDRESSABLE INTERNAL RAM              *
                 87      ;*************************************************************
----             88              DSEG    AT 25H  ;on-chip data segment in
0025             89      BUFFER: DS      COUNT   ; in bit-addressable space
                 90              END
```

图11-16 （续）

11.8.2 使用8255

8255是一种可编程外围接口芯片（PPI），可用于扩展8051的I/O口。8255有40个引脚，3个8位的I/O端口：端口A，端口B和端口C。8255也有1个低电平有效的输入控制信号$\overline{RD}$和$\overline{RW}$，使用时可直接连接到8051的$\overline{RD}$和$\overline{RW}$。同时，D0~D7数据引脚可连接到8051的数据总线。

A0和A1是2个输入控制引脚，可用于选择8255的相应端口（见表11-2）。但需要注意的是，A0和A1均为高电平表示不选中任何端口，而是选择控制寄存器。8255内部的控制寄存器是1个8位寄存器（见表11-3），用于设定3个I/O端口的工作模式。这与8051的TMOD和SCON寄存器类似，可分别用于设置8051的定时器和串行端口

的工作模式。

表11-2　8255端口选择

A1	A0	选　择	A1	A0	选　择
0	0	端口*A*	1	0	端口*C*
0	1	端口*B*	1	1	控制寄存器

285 ~ 288

表11-3　8255控制寄存器

位	组	描　述
D7	A	1＝I/O模式 0＝BSR模式
D6	A	模式选择位1
D5	A	模式选择位0 00＝模式0 01＝模式1 10＝模式2 11＝模式3
D4	A	端口A 1＝输入 0＝输出
D3	A	端口C（高4位PC7~PC4） 1＝输入 0＝输出
D2	B	模式选择位 0＝模式0 1＝模式1
D1	B	端口B 1＝输入 0＝输出
D0	B	端口C（低4位PC3~PC0） 1＝输入 0＝输出

8255的端口可被设置工作于上述4个模式之一，这里只讨论最简单的模式0（基本I/O）。该模式提供简单的类似于8051端口0到3的输入/输出操作。其他更复杂的模式可在两个器件之间（此处指8051和I/O设备之间）提供具有握手信号的智能通信功能。

例11-7　设计目标

编程实现以下目标：读取8个开关的状态（见图11-17），对每个闭合的开关点亮

其所对应的LED。

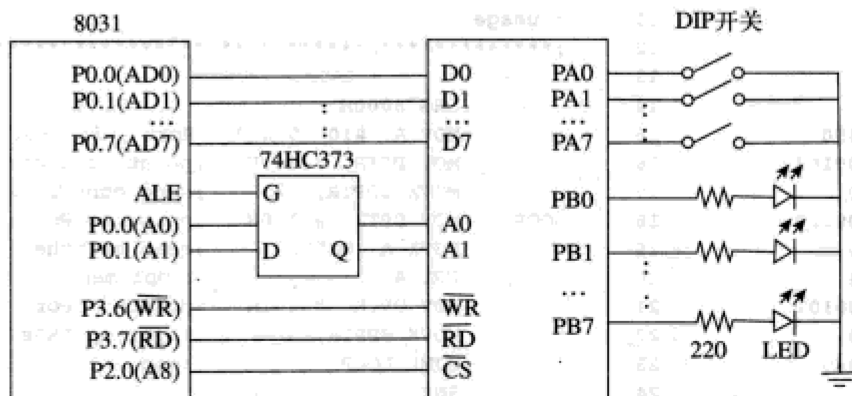


图11-17 8255接口

注意图11-17中，8255和8051的连接方式类似于一个外部存储器，这种I/O器件的连接概念也类似于存储器的“空间映射”，使得8051在访问8255的端口和控制寄存器时和访问外部存储器的方式相同。地址总线的2个最低有效位A1和A0直接连接到8255的A1和A0，用于设置选择8255的端口或控制寄存器，同时地址线A8连接到8255的片选信号（CS），所以地址总线的设置如下：

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
X	X	X	X	X	X	X	1	X	X	X	X	X	X	A1	A0

假设不确定位“X”均为0，那么地址0100H、0101H、0102H将分别选择端口*A*、*B*和*C*，而地址0103H选择控制寄存器。

289

这是一个很简单的接口扩展的例子，程序清单如图11-18所示。第1步为了设置端口的工作模式先选择控制寄存器，然后往控制寄存器里写用于将端口*A*设置为输入及将端口*B*设置为输出的恰当的控制字。可以看出，连接到端口*A*的开关和连接到端口*B*的LED一一对应起来。例如，当合上连接在A0引脚的开关时，将产生一个低电平信号，8051将B0引置为高电平，把与其相连的LED点亮。因此，首先选择端口*A*并将其内容（标示此时8个开关的状态）读入到累加器中，然后，将累加器的内容取反，再选择端口*B*，并将累加器的内容传送到端口*B*，即能完成上述任务。

```
1       $DEBUG
2       $NOSYMBOLS
3       $NOPAGING
4       ;FILE: 8255.SRC
5       ************************************************
6       ;          8255 PPI INTERFACE EXAMPLE          *
7       ;                                              *
8       ; This program uses the 8255 PPI to connect to *
```

图11-18 8255接口程序

```
                 9   ; 8 switches and 8 corresponding LEDs.  Using *
                10   ; the 8255 gives additional 8 I/O pins for I/O *
                11   ; usage.                                       *
                12   ;**********************************************
                13
8000            14          ORG 8000H
8000 7490       15          MOV A, #10010000B    ;Port A=in, Port B=out
8002 900103     16          MOV DPTR, #0103H     ;point to control reg
8005 F0         17          MOVX @DPTR, A        ;send control word
8006 900100     18   LOOP:  MOV DPTR, #0100H     ;point to Port A
8009 E0         19          MOVX A, @DPTR        ;read switches
800A F4         20          CPL A                ;complement
800B 900101     21          MOV DPTR, #0101H     ;point to Port B
800E F0         22          MOVX @DPTR, A        ;light up LEDs
800F 80F5       23          SJMP LOOP            ;loop
                24          END
```

290

图11-18 (续)

11.9 RS232（EIA-232）串行接口

我们已经知道，8051包含一个与串行I/O设备相连的片内串行端口。实际上，也可以将8051的串行端口和PC机的串行端口连接起来进行通信。PC机串口也同样遵循RS232 (EIA232) 协议标准，所以可以采用标准的RS232电缆连接PC和8051，RS232电缆的两端一般为DB-25连接器。但由于DB-25的有些引脚在串行通信的过程中是闲置不用的，所以另外一种只有9根线的DB-9型连接器更为常用。无论是DB-25还是DB-9，最重要的都是3根线，即接收数据线 (RXD) 、发送数据线 (TXD) 和地线 (GND) 。RS232串行接口允许采用握手协议，即为了建立通信信道，一个设备向另外一个设备发送请求发送数据 (RTS) 信号，同时等待该设备回传清除发送 (CTS) 信号。一旦接收到CTS信号，这两个设备之间就可以通信了。

当将8051连接到PC机的RS232进行串行通信时，需要解决二者之间的电平匹配问题。8051采用的是TTL电平，即5V表示高，0V表示低。但对RS232协议来讲，+3~+15V为高，－5~15V为低。所以，在8051和RS232进行连接时，需要添加一种电平转换器件。该器件的基本功能是实现两种不同电平间的相互转换，也就是说使得RS232的“高”和“低”转变成能为8051理解的“高”和“低”，反之亦然。图11-19展示了在8051和RS232连接时采用1488/1489作为电平转换器的连接电路图。

291

例11-8 设计目标

8051通过RS232串行接口连接到PC机，编程实现，从PC机输入十进制数字到8051串行端口，8051再在接收的数字回传到PC机，然后在显示器上显示出来。

程序清单如图11-20所示，首先调用FACE子例程，初始化串行端口，并在传送

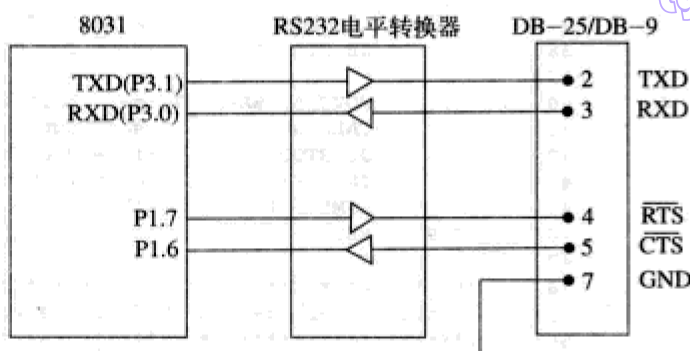


图11-19 RS232 接口电路图

信息头之前完成握手操作。为了实现设计目标，将8051的串行端口设置为工作方式1，即8位UART，波特率为9600。在发送完$\overline{\text{RTS}}$信号之后，8051将等待从串行端口传来对方的$\overline{\text{CTS}}$信号。只有在探测到$\overline{\text{CTS}}$信号后，再通过串行端口发送初始信息。

```
                         1       $DEBUG
                         2       $NOSYMBOLS
                         3       $NOPAGING
                         4       ;FILE: SERIAL.SRC
                         5       ;*************************************************
                         6       ;          RS232 SERIAL INTERFACE EXAMPLE        *
                         7       ;                                                *
                         8       ; This program obtains a decimal number from     *
                         9       ; the PC through the RS232 serial interface      *
                        10       ; and outputs the corresponding ASCII code to    *
                        11       ; the screen.                                    *
                        12       ;*************************************************
                        13
  020E                  14       INCHAR   EQU      020EH
  01DE                  15       OUTCHR   EQU      01DEH
  0097                  16       RTS      EQU      P1.7
  0096                  17       CTS      EQU      P1.6
                        18
8000                    19                ORG 8000H
8000 12800F             20                CALL FACE          ;initial & handshake
8003 908043             21                MOV DPTR, #ASC     ;point to ASCII table
8006 12020E             22       LOOP:    CALL INCHAR        ;get decimal number
8009 93                 23                MOVC A, @A+DPTR    ;convert to ASCII
800A 1201DE             24                CALL OUTCHR        ;output ASCII
800D 80F7               25                SJMP LOOP          ;loop
                        26
                        27       ;*************************************************
                        28       ; Subroutine for initialization & handshaking   *
                        29       ;*************************************************
                        30
800F 758920             31       FACE:    MOV TMOD, #20H   ;timer 1, mode 2
8012 758D98             32                MOV TH1, #98H    ;reload count for
                                                             9600baud
8015 759852             33                MOV SCON, #52H   ;serial port, mode 1
8018 90802C             34                MOV DPTR, #MSG   ;pointer to initial msg
801B D28E               35                SETB TR1         ;start timer 1
                        36
```

图11-20 RS232接口程序

```
801D C297        37              CLR RTS           ;assert RTS (handshaking)
801F 2096FD      38              JB CTS, $         ;wait for CTS
                 39
8022 93          40     OUT:     MOVC A, @A+DPTR   ;get characters
8023 1201DE      41              CALL OUTCHR       ;send out
8026 6003        42              JZ STOP           ;if end of message, stop
8028 A3          43              INC DPTR          ;else, get next character
8029 80F7        44              SJMP OUT          ;loop
802B 22          46     STOP:    RET
                 47
                 48     ;*************************************************
                 49     ; Initial message and ASCII codes for numbers   *
                 50     ;*************************************************
                 51
                 52     ;initial message
802C 504C4541    53     MSG:     DB 'PLEASE ENTER NUMBER = ', 00H
8030 53452045
8034 4E544552
8038 204E554D
803C 42455220
8040 3D20
8042 00
                 54
                 55     ;ASCII codes for decimal numbers
8043 30          56     ASC:     DB 30H   ;0
8044 31          57              DB 31H   ;1
8045 32          58              DB 32H   ;2
8046 33          59              DB 33H   ;3
8047 34          60              DB 34H   ;4
8048 35          61              DB 35H   ;5
8049 36          62              DB 36H   ;6
804A 37          63              DB 37H   ;7
804B 38          64              DB 38H   ;8
804C 39          65              DB 39H   ;9
                 66              END
```

图11-20 （续）

然后程序将调用INCHAR子例程，等待从出口传送来的十进制数字。一旦探测到有数字输入，将从查找表中检索出相应的ASCII码，之后调用OUTCHAR子例程，把该ASCII码通过串行端口发送出去。

292 ~ 293

11.10 CENTRONICS并行接口

CENTRONICS并行接口是应用最广泛的打印机接口。实际上其已经成为并行接口的标准，地位的重要性类似于前一节讨论的RS232。CENTRONICS并行接口定义了36条信号线。具体的物理实现可以选择36针的CENTRONICS连接器，或者DB-25连接器（见表11-4），目前PC机附加的都是DB-25连接器接口，所以后者的应用范围更广泛。本节将展示8051如何通过DB-25连接器连接到外部打印机。

CENTRONICS并行接口包括8条数据线，用于并行传送1个字节的数据到打印机。当待打印的数据字节已经放置到数据总线上时，8051将产生1个$\overline{\text{STROBE}}$信号，

通知打印机数据已经存在于数据总线上，如果打印机此时处于空闲状态，就向8051回应1个应答信号 $\overline{\text{ACK}}$，表示自己已经准备完毕，可以处理8051的打印请求。否则，如果打印机此时处于忙碌状态，就向8051回传1个BUSY信号，表示不能接受其打印请求。

表11-4 DB-25 CENTRONICS并行接口引脚定义

引　脚	描　述	引　脚	描　述
1	$\overline{\text{STROBE}}$	10	$\overline{\text{ACK}}$（应答）
2	DATA0	11	BUSY
3	DATA1	12	PAPER END
4	DATA2	13	SLCT（选择）
5	DATA3	14	$\overline{\text{AUTOFEED}}$
6	DATA4	15	$\overline{\text{ERROR}}$
7	DATA5	16	INITIALIZE PRINTER
8	DATA6	17	$\overline{\text{SLCTIN}}$（在……中选择）
9	DATA7	18~25	GND（地）

例11-9　设计目标

编程实现，查询打印机的当前状态，并连续的发送测试信息到打印机。

图11-21是8051通过CENTRONICS并行接口连接到打印机的电路图。相应的控制软件清单如图11-22所示。该程序首先产生 $\overline{\text{STROBE}}$ 信号，之后不断检测打印机直道其传回空闲信号（BUSY＝0），而且纸还未走完（PAPER END＝0）；SLCT信 294

8031　　CENTRONICS 接口

P1.7 — 2 DATA7
P1.6 — 3 DATA6
P1.5
P1.4
P1.3
P1.2
P1.1 — 8 DATA1
P1.0 — 9 DATA0
P3.0 — 1 $\overline{\text{STROBE}}$
P3.1 — 10 $\overline{\text{ACK}}$
P3.2 — 11 BUSY
P3.3 — 12 PAPER END
P3.4 — 13 SLCT
P3.5 — 15 $\overline{\text{ERROR}}$
18 GND

图 11-21

号为高电平；没有错误发生（$\overline{ERROR}=1$）。如果上述任何1个条件不满足，将给出错误信息并结束程序。8051将等待打印机的应答信号 $\overline{ACK}$，一旦该信号有效，则将从测试信息得到的字符通过P1口发送到打印机。

```
                1       $DEBUG
                2       $NOSYMBOLS
                3       $NOPAGING
                4       ;FILE: PARALLEL.SRC
                5       ;*************************************************
                6       ;     CENTRONICS PARALLEL INTERFACE EXAMPLE     *
                7       ;                                               *
                8       ; This program continually sends a test message *
                9       ; to the printer.                               *
               10       ;*************************************************
               11
  0090         12       DAT     EQU   P1
  00B0         13       STR     EQU   P3.0
  00B1         14       ACK     EQU   P3.1
  00B2         15       BUSY    EQU   P3.2
  003C         16       MASK    EQU   00111100B
  0030         17       OK      EQU   00110000B
               18
8000           19               ORG 8000H
8000 90801C    20       STRT:   MOV DPTR, #MSG   ;point to test message
8003 75B03C    21       LOOP:   MOV P3, #MASK    ;activate STROBE, and
               22                                ; P3.1-P3.5 = input
8006 E5B0      23               MOV A, P3        ;read printer status
8008 543C      24               ANL A, #MASK     ;only P3.1-P3.5
800A B4300D    25               CJNE A, #OK, ERR ;any error?
800D E4        26       SEND:   CLR A            ;no: get ready to send
800E 20B1FD    27               JB ACK, $        ;before send, wait for ACK
8011 93        28               MOVC A, @A+DPTR  ;get char in test msg
8012 F590      29               MOV DAT, A       ;send char to printer
8014 A3        30               INC DPTR         ;point to next char
8015 B400EB    31               CJNE A, #00H, LOOP ;not end of msg? loop
8018 80E6      32               SJMP STRT        ;end of msg, back to start
               33
801A 801F      34       ERR:    SJMP STOP        ;printer error, stop
               35
               36       ;*************************************************
               37       ; Test message for the printer                  *
               38       ;*************************************************
               39
               40       ;test message
801C 54484953  41       MSG:    DB 'THIS IS A TEST FOR THE PRINTER', 00H
8020 20495320
8024 41205445
8028 53542046
802C 4F522054
8030 48452050
8034 52494E54
8038 4552
803A 00
803B 00        42       STOP:   NOP
               43               END
```

图11-22 CENTRONICS并行接口程序

11.11 模拟输出

利用微控制器测控外界物理量时，经常需要产生或感知模拟信号，这对于微控制器来说是比较容易的事情。本设计例子使用了2个电阻、2个电容、1个电位器、1片LM301 OP放大器和1片MC1408L8数模转换器（8位），这两片IC都是比较便宜和常用的。通过8051的端口1将8位数据传送到数模转换芯片（如图11-23所示）。在按图搭建好电路后连接到SBC-51，即可以使用监控命令进行测试了。当向端口1写入不同的数值并调整1K电位器时，测量LM301第6脚的输出电压（单位：V）。该电压可以从0V（P1＝00H）变化到10V（P1＝FFH）。

295
~
296

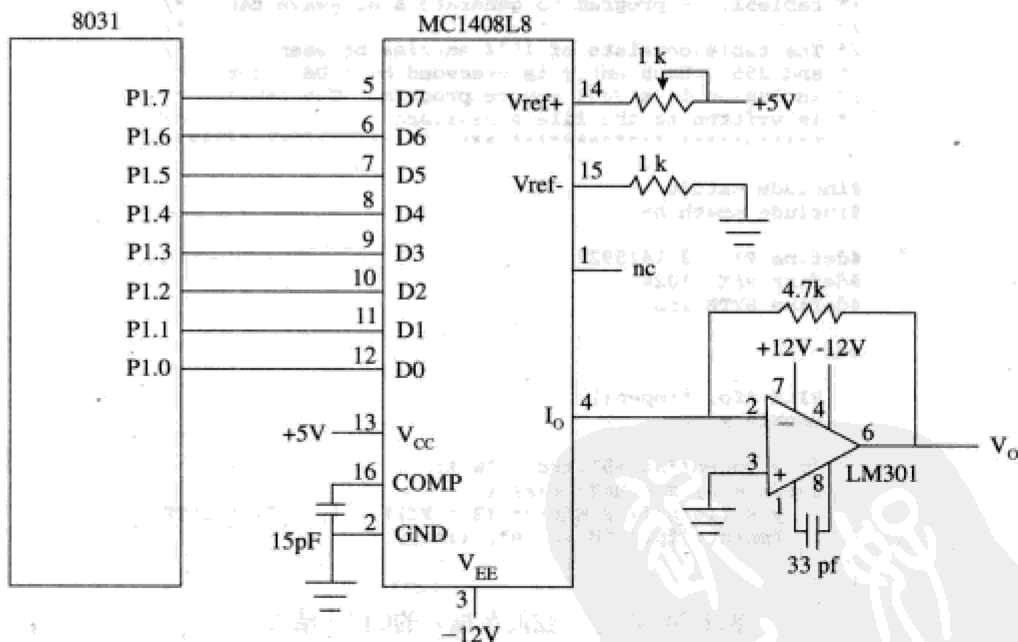


图11-23 MC1408L8的接口电路图

当硬件电路完成后，就应该开始着手准备编写接口软件了。常用的测试程序是令其产生一个锯齿波信号（通过将相应值传送到D/A转换芯片的方式实现）。但是，目前在这里打算利用DAC产生1个正弦信号，这当然是一个具有挑战性的问题。

例11-10 设计目标

编程实现，采用图11-23的DAC接口电路产生1个正弦信号，用常数STEP来设置正弦波的频率。另外要求在程序中利用中断产生10kHz的刷新频率（即每隔100μs更新1次DAC数值）。

由于8051处理数据的能力是有限的，所以唯一可行的方案是采用查表法来完成这个任务。于是现在就需要1个和单周期正弦信号对应的8bit的数值构成的数据表。

该数据表以127数值开始，逐渐增减到255，然后再逐次递减到0（中间经过127），而后再逐渐增加到127，即按正弦图线的规律变化。

现在的问题是需要1个很大的描述正弦波信号的数据表，但如何产生这个数据表呢？手工的办法肯定是不可行的。最容易的办法是，采用其他高级语言来建立该数据表并将全部数据存成1个文件，然后再将其导入到8051源程序里。图11-24中的table.51 C语言程序可以完成该工作。该程序产生具有1024个采样点的正弦信号数据表，且每个数据值限于0~255之间。该程序会产生1个输出文件sine51.src，为了同8051的源代码兼容，每项数据都以"DB"开头。

```
/**********************************************************/
/* table51.c - program to generate a sinewave table */
/*                                                  */
/* The table consists of 1024 entries between 0     */
/* and 255.  Each entry is preceded by ' DB ' for   */
/* inclusion in a 8051 source program.  The table   */
/* is written to the file sine51.src                */
/**********************************************************/

#include <stdio.h>
#include <math.h>

#define PI   3.1415927
#define MAX  1024
#define BYTE 255

main()
{
   FILE *fp, *fopen();
   double x, y;

   fp = fopen("sine51.src", "w");
   for(x = 0; x < MAX; ++x) {
      y = ((sin((x / MAX) * (2 * PI)) + 1) / 2) * BYTE;
      fprintf(fp," DB %3d\n", (int)y);
   }
}
```

图11-24 产生正弦波数据表的C程序清单

8051产生正弦波的程序清单如图11-25所示。其主循环（36~40行）完成3件事情：将定时器0初始化为每100μs中断1次；允许中断；无限循环。定时器0中断服务子例程（41~51行）完成下面这些工作，每隔100μs采用查表法从正弦波数据表中读取1个数值，而后通过DPTR将该数值写到8051的端口1。常数STEP决定着相邻两次从数据表中读取数据时间隔的元素个数（将数据表视为一维数组），STEP定义在程序的第26行，只用内部RAM的1个字节，其初始化需要通过监控命令来完成。在每次调用ISR时将DPTR和STEP的值求和得到数值作为下一次从数据表里取数据地址。在第69行采用ORG指令将数据表的位置设定在8400H，即开始于偶数个1K字节的存储页。如果DPTR增加到87FFH（即数据表的结尾）后，再回到初始地址。由于数据表较大，所以采用$NOLIST汇编伪指令（77行）通知汇编器，不要将其后的数据输出到列表文件中。在1092行（未显示）用$LIST伪指令通知汇编器，可以把此后的内容输出到列表文件。正弦波信号的频率由3个参数来确定：STEP，数据表的大

小，定时器的中断周期，具体细节请参阅程序清单的16~20行。

```
                      1  $debug
                      2  $nopaging
                      3  $nosymbols
                      4  ;FILE: DAC.SRC
                      5  ;*************************************************************
                      6  ;            MC1408L8 INTERFACE EXAMPLE                      *
                      7  ;                                                            *
                      8  ; This program generates a sine wave using a sine            *
                      9  ; wave look-up table and an interface to a MC1408L8          *
                     10  ; 8-bit digital-to-analog converter.  The program is        *
                     11  ; interrupt-driven.                                          *
                     12  ;                                                            *
                     13  ; Data are read from a 1024-entry sine wave table and        *
                     14  ; sent to the DAC every 100 us.  Each value sent is          *
                     15  ; STEP locations past the previous value sent (with          *
                     16  ; wrap around once the end is reached).  The period          *
                     17  ; of the sine wave is 100 x (1024 / STEP) us.  For           *
                     18  ; example, if STEP is 20H, the sine wave has a period        *
                     19  ; of 100 x (1024 / 32) = 3.2 ms and a frequency of           *
                     20  ; 313 Hz.                                                    *
                     21  ;                                                            *
                     22  ; Note:  Initialize STEP in internal location 50H            *
                     23  ; before running the program.                                *
                     24  ;*************************************************************
  00BC               25  MONITOR  CODE   00BCH        ;MON51 entry (V12)
  0050               26  STEP     DATA   50H          ;put STEP in internal RAM
                     27
8000                 28           ORG    8000H        ;start at 8000H
8000 028015          29           LJMP   MAIN         ;initialize timer
8003 028037          30           LJMP   EXT0ISR      ;unsused
8006 028022          31           LJMP   T0ISR        ;every 100 us, update DAC
8009 028037          32           LJMP   EXT1ISR      ;unused
800C 028037          33           LJMP   T1ISR        ;unused
800F 028037          34           LJMP   SPISR        ;unused
8012 028037          35           LJMP   T2ISR        ;unused
8015 758902          36  MAIN:    MOV    TMOD,#02H    ;8-bit, auto reload
8018 758C9C          37           MOV    TH0,#-100    ;100 us delay
801B D28C            38           SETB   TR0          ;start timer
801D 75A882          39           MOV    IE,#82H      ;enable timer 0 interrupts
8020 80FE            40           SJMP   $            ;main loop does nothing!
8022 E550            41  T0ISR:   MOV    A,STEP       ;add STEP to DPTR
8024 2582            42           ADD    A,DPL
8026 F582            43           MOV    DPL,A
8028 5002            44           JNC    SKIP
802A 0583            45           INC    DPH
802C 538303          46  SKIP:    ANL    DPH,#03H     ;wrap around, if necessary
802F 438384          47           ORL    DPH,#HIGH(TABLE)
8032 E4              48           CLR    A
8033 93              49           MOVC   A,@A+DPTR    ;get entry
8034 F590            50           MOV    P1,A         ;send it
8036 32              51           RETI
                     52
                     53  EXT0ISR:                     ;unused interrupts
                     54  EXT1ISR:
                     55  T1ISR:
                     56  T2ISR:
8037 C2AF            57  SPISR:   CLR    EA           ;turn off interrupts and
8039 0200BC          58           LJMP   MONITOR      ; return to MON51
                     59
```

图11-25　8051产生正弦波的程序清单

```
                60      ;*************************************************************
                61      ; The following is a sine wave look-up table.  The          *
                62      ; table contains 1024 entries and is ORGed to begin        *
                63      ; at 8400H to allow easy wrap-around of the DPTR           *
                64      ; once the end of the table is reached.  The entries       *
                65      ; are 8-bits each (0 to 255) for output to an 8-bit        *
                66      ; DAC.  The table was generate from a C program and        *
                67      ; read into this 8051 program.                             *
                68      ;*************************************************************
8400            69                ORG     8400H
8400 7F         70      TABLE:    DB      127
8401 80         71                DB      128
8402 81         72                DB      129
8403 81         73                DB      129
8404 82         74                DB      130
                75      ; Listing turned off after first five entries
                76      ;------------------------------------------------------------
                77 +1   $NOLIST
              1093      ; Listing turned back on for last five entries
87FB 7B       1094                DB      123
87FC 7C       1095                DB      124
87FD 7D       1096                DB      125
87FE 7D       1097                DB      125
87FF 7E       1098                DB      126
              1099                END
```

297
≀
299

图11-25 （续）

11.12 模拟输入

本节将讨论1个模拟输入的设计例子，电路如图11-26所示，主要包括1个电阻、1个电容、1个电位器和1片A/D转换芯片ADC0804。ADC0804是比较便宜的ADC芯

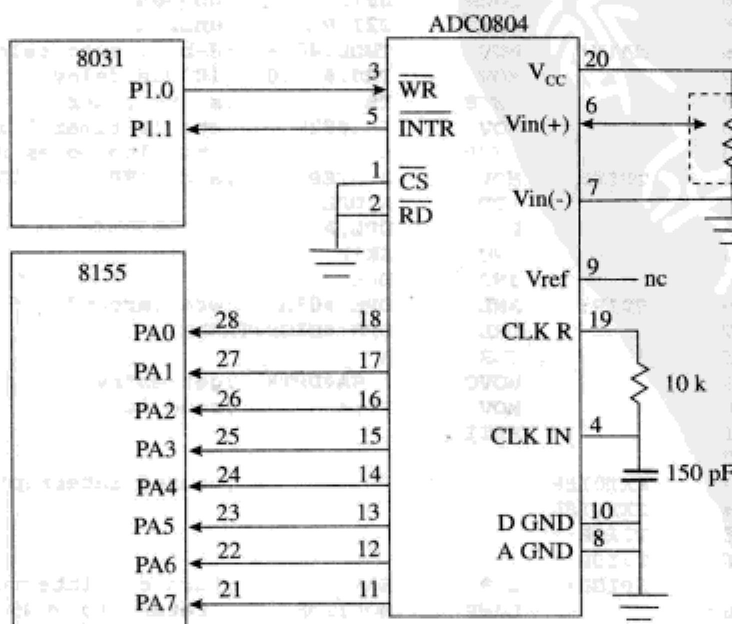


图11-26 ADC0804接口图

片[美国国家半导体(NS)公司的产品]，可以在约100μs的时间里将输入电压转换成8位的数字量。

ADC0804由写输入（$\overline{WR}$）和中断输出（$\overline{INTR}$）来控制，将$\overline{WR}$置为低电平即启动A/D转换操作。当转换完成时（约100μs之后），ADC0804会将$\overline{INTR}$信号置低，表示A/D转换已经完成。直到启动下一次转换时（即$\overline{WR}$由1变到0，下降沿），才将$\overline{INTR}$信号恢复为高电平。（$\overline{WR}$）和（$\overline{INTR}$）分别连接到8051的P1.1和P1.0。如图所示，在本例中使用8155的端口A完成数据传输。

ADC0804的内部时钟频率由接在其19脚和4脚的RC网络来确定。输入电压信号采用的是差分形式，分别接到Vin（+）（6脚）和Vin（−）（7脚）。在本例中，Vin（−）直接接地，Vin（+）接到电位器的中心抽头。Vin（+）的输入电压范围为0~5V，具体由电位器来调控。关于ADC0804的更详细的内容请参阅其产品数据表。

例11-11　设计实例

编写程序完成，连续监测电位器中心抽头的电压（通过ADC0804芯片实现A/D转换）。并将结果以ASCII字符的形式输出到控制台上。

图11-27是完成上述功能的程序清单。由于8155的端口在复位后即默认为输入状态，所以不需要再进行初始化了。端口A对应于外部存储器的0101H地址，可方便的使用MOVX指令来读取其内容。将ADC0804的$\overline{WR}$（连接在8051的P1.0）先清零再置1（34~35行），即可启动A/D转换过程。之后程序进入1个等待循环，一直到ADC0804完成本次转换并将$\overline{INTR}$（连接到P1.1）信号清零（36行）。在程序的第37行和第38行，读取A/D转换结果，并利用MON51的OUT2HX子例程将结果发送到控制台（39行）。当该程序正常运行时，1个字节的数据将显示在控制台上。随着电位器的调整，结果的变化范围为00H~FFH。

300
~
301

```
1       $debug
2       $nopaging
3       $nosymbols
4       ;FILE: ADC.SRC
5       ;*******************************************************
6       ;               ADC0804 INTERFACE EXAMPLE              *
7       ;                                                      *
8       ; This program reads analog input data from an         *
9       ; ADC0804 interfaced to Port A of the 8155.  The       *
10      ; result is reported on the console as a hexadecimal   *
11      ; byte.  The following steps occur:                    *
12      ;                                                      *
13      ;        1. Send banner message to console             *
14      ;        2. Toggle ADC0804 -WR line (P1.0) to begin    *
15      ;           conversion                                 *
16      ;        3. Wait for ADC8084 -INTR line (P1.1) to go   *
17      ;           low indicating "conversion complete"       *
18      ;        4. Read data from 8155 Port A                 *
19      ;        5. Output data to console in hexadecimal      *
20      ;        6. Go to step 2                               *
21      ;*******************************************************
22      PORTA    EQU      0101H        ;8155 Port A
```

图11-27　ADC0804接口例程清单

```
23   CR       EQU    0DH            ;ASCII carriage return
24   LF       EQU    0AH            ;ASCII line feed
25   ESC      EQU    1BH            ;ASCII escape
26   OUT2HX   EQU    028DH          ;MON51 subroutine
27   OUTSTR   EQU    0282H          ;MON51 subroutine
28   WRITE    BIT    P1.0           ;ADC0804 -WR line
29   INTR     BIT    P1.1           ;ADC0804 -INTR line
30
31            ORG    8000H
32   ADC:     MOV    DPTR,#BANNER   ;send message
33            CALL   OUTSTR
34   LOOP:    CLR    WRITE          ;toggle -WR line
35            SETB   WRITE
36            JB     INTR,$         ;wait for -INTR = 0
37            MOV    DPTR,#PORTA    ;init DPTR --> Port A
38            MOVX   A,@DPTR        ;read ADC0804 data
39            CALL   OUT2HX         ;send data to console
40            MOV    DPTR,#LEFT2    ;back-up cursor by 2
41            CALL   OUTSTR
42            SJMP   LOOP           ;repeat
43
44   BANNER:  DB     '*** TEST ADC0804 ***',CR,0


45   LEFT2:   DB     ESC,'[2D',0    ;VT100 escape sequence

46            END
```

图11-27　（续）

图11-27的程序清单测量的是模拟电压信号的近似值，如果将电位器替换为其他模拟输入器件也是可行的。例如，在温度测量过程中可以选用热敏电阻（一种阻值随环境温度变化的器件）；如果要输入语音信号，需要使用麦克风作为模拟信号的输入器件。ADC0804的转换周期为100μs，即采样频率为10kHz。根据采样定理，可用ADC0804来处理带宽小于5kHz的信号（大约是保证电话语音质量所需的带宽）。为了将典型麦克风的微弱信号放大为可以被ADC0804接受的0~5V信号，需要在系统中附加完成放大功能的相应电路。另外，还需要1个采样－保持电路，使输入信号的电压值在A/D转换期间维持恒定，关于这部分电路，读者可自行考虑如何实现。

302

11.13　传感器的接口

在第6章曾经讨论过1个利用简单的温度传感器控制锅炉温度的例子。此处将考虑1个更复杂的温度传感器，即DS1620数字温度计和调温器芯片，由MAXIM公司生产①。

DS1620是1款有8个引脚的集成芯片，具有3个温度报警输出信号：T_{HIGH}，T_{LOW}和T_{COM}。当芯片测量到的温度大于或等于由用户设定的上限TH时，T_{HIGH}将变为高

① Maxim Integrated Products, Inc., 120 San Gabriel Drive, Sunnyvale, CA 94086。

电平；当温度小于或等于用户设定的下限TL时，T_{LOW}将变为高电平。另外，当温度超过TH时T_{COM}变成高电平，并一直保持到温度下降到TL以下才变为低电平。

DS1620芯片和8051相连接时，采用3线通信方式，因为在实际的通信过程中也确实是通过3条线来完成的，即数据输入/输出（DQ），时钟输入（CLK/CONV）和复位（RST）。将RST信号置1即可启动通信过程，时钟信号通过CLK引脚输入到DS1620，而数据将通过DQ引脚被写入或读出，最低有效位在先。

图11-28描述了8051和DS1620的连接电路。8051和DS1620通信需要使用表11-5列出的命令来实现。

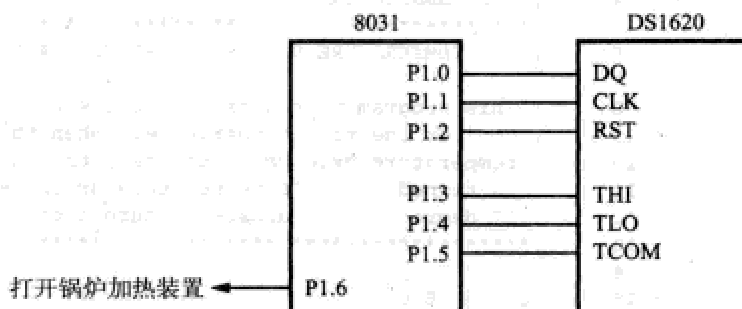


图11-28 DS1620的接口电路

表11-5 DS1620指令集

代 码	指 令	描 述
AAH	读温度	从温度寄存器读取最后一次温度转换结果
A0H	读计数器	读取计数器的剩余数值
A9H	读斜率	读取斜率累加器的数值
EEH	开始转换	启动温度转换
22H	结束转换	结束温度转换
01H	写TH	写TH值到TH寄存器
02H	写TL	写TL值到TL寄存器
A1H	读TH	从TH寄存器读取TH值
A2H	读TL	从TL寄存器读取TL值
0CH	写配置	写配置数据到配置寄存器
ACH	读配置	从配置寄存器读取当前的配置数据

例11-12 设计目标

编程实现，利用温度传感器将房间的温度保持在20°C附近。如果温度低于17°C则打开锅炉加热装置进行升温；反之如果温度高于23°C则关闭锅炉加热装置实现降温。

图11-29是完成上述设计任务的程序清单。首先关闭锅炉加热装置，并完成

DS1620的初始化，这需要先发1条“写配置”命令到DS1620的DQ引脚，然后接着发送1字节的配置代码到配置/状态寄存器（见表11-6）。在本例中，将配置码设为：CPU使用位为1，即DS1620和8051之间采用3线通信操作模式工作。同时，把1SHOT位清零，选择连续温度转换方式。需要注意的是，发送命令和配置码都是通过SEND子例程来完成的，一位接一位的发送，其最低有效位在先，发送到DQ。发送指令的过程是把RST信号置1来启动的，待指令发送完毕，清零RST，即停止传送。

```
                 1    $DEBUG
                 2    $NOSYMBOLS
                 3    $NOPAGING
                 4    ;FILE: SENSOR.SRC
                 5    ;*********************************************
                 6    ;     TEMPERATURE SENSOR INTERFACE EXAMPLE    *
                 7    ;                                             *
                 8    ; This program uses a temperature sensor to   *
                 9    ; monitor the room temperature.  When the     *
                10    ; temperature exceeds 23 degrees, the furnace *
                11    ; is turned off.  If temperature drops below  *
                12    ; 17 degrees, the furnace is turned on.       *
                13    ;*********************************************
                14
  0090          15    DQ      EQU     P1.0
  0091          16    CLK     EQU     P1.1
  0092          17    RST     EQU     P1.2
                18
  0093          19    THI     EQU     P1.3
  0094          20    TLO     EQU     P1.4
  0095          21    TCOM    EQU     P1.5
                22
  0096          23    FURN    EQU     P1.6
                24
8000            25            ORG 8000H
8000 C296       26            CLR FURN               ;turn furnace off
                27
8002 D292       28    CONF:   SETB RST               ;initiate transfer
8004 740C       29            MOV A, #0CH            ;write config
8006 128045     30            CALL SEND              ;send to DS1620
8009 740A       31            MOV A, #00001010B      ;CPU = 1; 1-shot=0
800B 128045     32            CALL SEND          ;send to DS1620
800E C292       33            CLR RST            ;stop transfer
                34
8010 D292       35            SETB RST           ;initiate transfer
8012 7401       36            MOV A, #01H        ;write TH
8014 128045     37            CALL SEND          ;send to DS1620
8017 7430       38            MOV A, #48         ;TH = 48 x 0.5 = 24 deg C
8019 128045     39            CALL SEND          ;send to DS1620
801C C292       40            CLR RST            ;stop transfer
                41
801E D292       42            SETB RST           ;initiate transfer
8020 7402       43            MOV A, #02H        ;write TL
8022 128045     44            CALL SEND          ;send to DS1620
8025 7420       45            MOV A, #32         ;TH = 32 x 0.5 = 16 deg C
8027 128045     46            CALL SEND          ;send to DS1620
802A C292       47            CLR RST            ;stop transfer
                48
```

图11-29 DS1620接口程序清单

```
802C D292        49    CONV:   SETB RST        ;initiate transfer
802E 74EE        50           MOV A, #0EEH     ;start temperature
                                                sensing
8030 128045      51            CALL SEND       ;send to DS1620
8033 C292        52            CLR RST         ;stop transfer
                 53
8035 209305      54    SENS:   JB THI, OFF     ;if T >= 24 degrees, off
8038 209406      55            JB TLO, ON      ;if T <= 16 degrees, on
803B 80F8        56            SJMP SENS       ;loop
                 57
803D C296        58    OFF:    CLR FURN        ;turn furnace off
803F 80F4        59            SJMP SENS       ;keep sensing
                 60
8041 D296        61    ON:     SETB FURN       ;turn furnace on
8043 80F0        62            SJMP SENS       ;keep sensing
                 64    ;*********************************************
                 65    ; This subroutine sends a byte of command or  *
                 66    ; data to the DS1620.                         *
                 67    ;*********************************************
                 68
8045 7808        69    SEND:   MOV R0, #08     ;use R0 as counter
8047 C291        70    NEXT:   CLR CLK         ;start clock cycle
8049 13          71            RRC A           ;rotate A into C, LSB 1st
804A 9290        72            MOV DQ, C       ;send out bit to DQ
804C D291        73            SETB CLK        ;complete the clock cycle
804E D8F7        74            DJNZ R0, NEXT   ;process next bit
8050 22          75            RET
                 76
                 77            END
```

图11-29 （续）

表11-6 DS1620 配置/状态寄存器

位	名 称	描 述
7	DONE	转换完成标志位 1=转换完毕 0=转换进行中
6	THF	温度超上限标志位 1=温度≥TH
5	TLF	温度超下限标志位 1=温度≤TH
4	NVB	非易失存储器忙标志 1=写存储器操作正在进行中 0=存储器空闲
3	1	—
2	0	—
1	CPU	CPU使用标志位 0=独立工作 1=在CPU控制下工作，即3线通信模式
0	1SHOT	温度测量方式标志位 1=单次转换方式，即接收到“开始转换”命令即转换1次 0=连续转换方式

在“写配置”命令之后，需要用户设置温度上限TH和温度下限TL。接着，发送“开始转换”命令启动温度测量过程。然后，8051进入1个无限循环里，不断地检查T_{HIGH}和T_{LOW}报警信号，并分别做出关闭和打开锅炉加热装置的应对措施。

11.14 继电器接口

继电器是一种触点的打开和关闭由外加电流驱动磁心产生的磁场来控制的开关。事实上，继电器被认为是电磁机械开关，可广泛应用于需要用电的通断来控制开关的开启/关闭的场合。

304 ~ 306

本例将选用OMRON[①]公司的单刀双掷型继电器G6RN来做1个项目，G6RN的内部结构如图11-30所示。

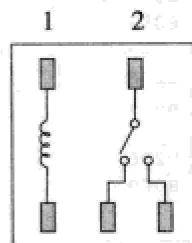


图11-30 G6RN的内部结构图

例11-13 设计目标

考虑一个简单的人行横道指示灯系统（如图11-31所示），要求应用8051和继电

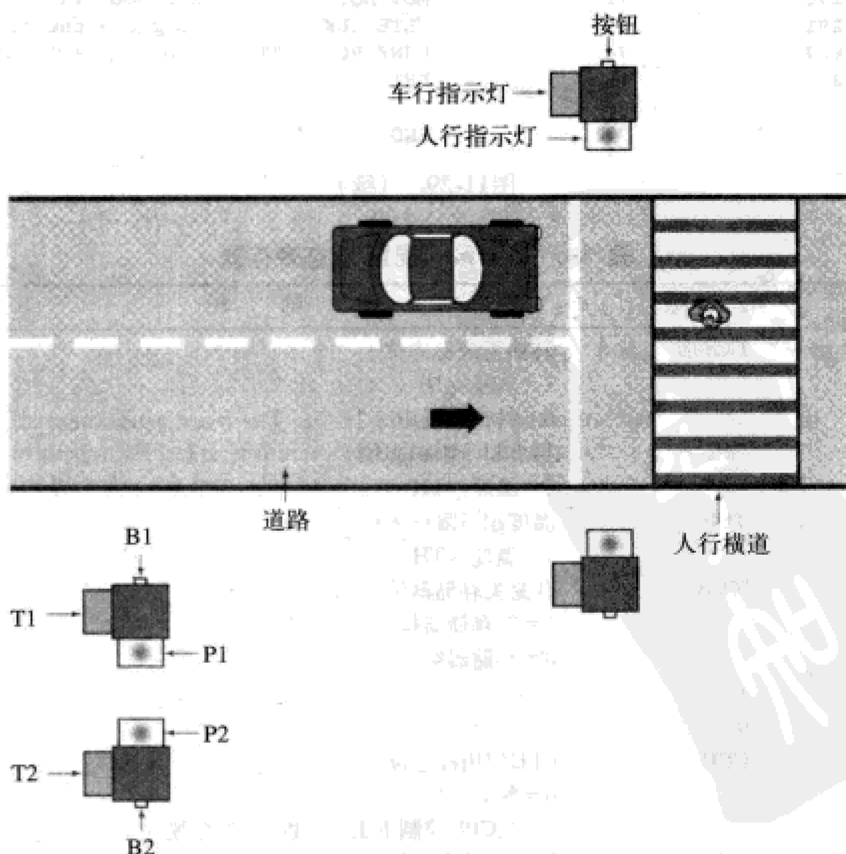


图11-31 行人横穿马路交通灯系统

① OMRON公司。

器设计这样一个交通灯系统。

图11-32给出了8051、继电器、行人按钮、交通指示LED的连接电路原理图。相应的程序清单如图11-33所示。主程序先允许后面要用到的相关中断，然后等待打算横穿马路的行人按下分布在道路两侧的按钮，即B1和B2。在初始状态，P1.0为低电平，继电器的开关状态是使得车行绿灯亮，人行红灯亮(即汽车此时可正常通行，而行人不允许横穿马路)。一旦有1个按钮（B1或B2）被按下，将对8051产生一个外部中断（$\overline{INT0}$）请求，然后程序会转到中断服务子例程EX0ISR去执行。在EX0ISR里将P1.0置为高电平，继电器的触点跳到另外一侧，使得车行红灯亮、人行绿灯亮，即汽车停车等候，行人可以横穿马路。这种状态将持续10s，然后又返回到车行绿灯亮、人行红灯亮的状态。

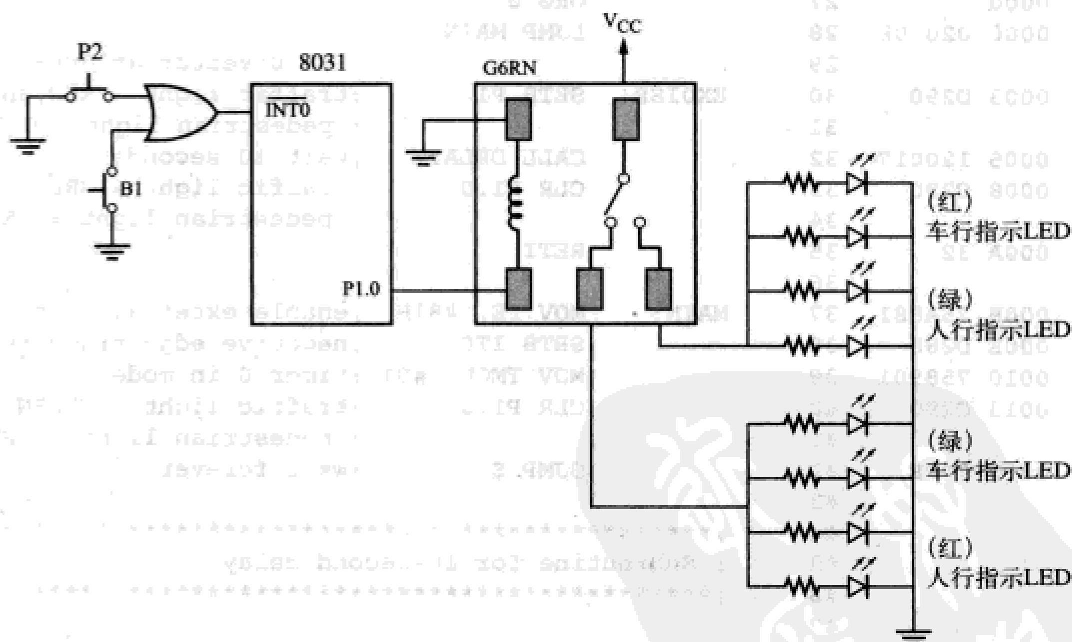


图11-32 行人横穿马路交通灯系统中继电器G6RN的接口电路图

```
1       $DEBUG
2       $NOSYMBOLS
3       $NOPAGING
4       ;FILE: STEPPER.SRC
5       ;*****************************************************
6       ; RELAY INTERFACE EXAMPLE:                           *
7       ;                   PEDESTRIAN TRAFFIC LIGHT         *
8       ;                                                    *
9       ; This program interacts with a relay to             *
10      ; control some RED and GREEN LEDs in a               *
11      ; pedestrian traffic light system.                   *
12      ;                                                    *
```

图11-33 继电器G6RN 接口程序清单

```
                  13  ; Initially, the GREEN traffic light and RED    *
                  14  ; pedestrian light are on.  If a pedestrian-    *
                  15  ; crossing button is pressed, a high-to-low     *
                  16  ; transition is generated at INT0.  This        *
                  17  ; signals the RED traffic light and the GREEN   *
                  18  ; pedestrian light to turn on.  A delay of 10   *
                  19  ; seconds ensues before the GREEN traffic light*
                  20  ; and RED pedestrian light are turned back on. *
                  21  ;**********************************************
                  22
  000A            23  TEN       EQU 10
  0064            24  HUNDRED   EQU 100 ;10 x 100 x 10000 us = 10 secs
  D8F0            25  COUNT     EQU -10000
                  26
0000              27            ORG 0
0000 02000B       28            LJMP MAIN
                  29                            ;EXT 0 vector at 0003H
0003 D290         30  EX0ISR:   SETB P1.0       ;traffic light = RED,and
                  31                            ; pedestrian light = GREEN
0005 120017       32            CALL DELAY      ;wait 10 seconds
0008 C290         33            CLR P1.0        ;traffic light = GREEN,
                  34                            ; pedestrian light = RED
000A 32           35            RETI
                  36
000B 75A881       37  MAIN:     MOV IE, #81H    ;enable external 0 int
000E D288         38            SETB IT0        ;negative edge triggered
0010 758901       39            MOV TMOD, #01   ;timer 0 in mode 1
0013 C290         40            CLR P1.0        ;traffic light = GREEN,
                  41                            ; pedestrian light = RED
0015 80FE         42            SJMP $          ;wait forever
                  43
                  44  ;**********************************************
                  45  ; Subroutine for 10-second delay               *
                  46  ;**********************************************
                  47
0017 7E0A         48  DELAY:    MOV R6, #TEN
0019 7F64         49  AGAIN1:   MOV R7, #HUNDRED
001B 758CD8       50  AGAIN2:   MOV TH0, #HIGH COUNT
001E 758AF0       51            MOV TL0, #LOW COUNT
0021 D28C         52            SETB TR0
0023 308DFD       53  WAIT:     JNB TF0, $
0026 C28D         54            CLR TF0
0028 C28C         55            CLR TR0
002A DFEF         56            DJNZ R7, AGAIN2
002C DEEB         57            DJNZ R6, AGAIN1
002E 22           58            RET
                  59
                  60            END
```

图11-33 （续）

11.15 步进电机接口

许多设备例如点阵打印机和软盘驱动器等都会用到一种特殊的电机，即步进电机。步进电机和传统的直流电机相比，其转动状况是可以精确控制的。因此，步进电机可应用于机械部件需要精确定位的场合。

首先简单回顾步进电机的基本原理。步进电机包括一个移动部件，称为**转子**，一般是由永磁材料制作而成的。包围着转子而且固定不动的部分称为**定子**，一般由两个缠绕在磁芯上的电磁铁构成。有关定子更详细的构造请参考图11-34。

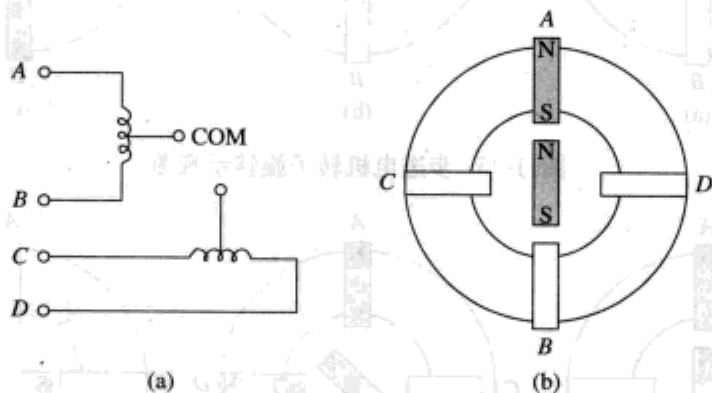


图11-34 步进电机定子线圈的结构

需要注意的是，两个线圈有一个中心抽头，即COM公共端，一般要连接到+5V。分析图11-34 a中垂直方向分布的两个线圈，两端分别被标记为*A*和*B*。垂直方向的这两个线圈缠绕在定子的上下两个铁芯上（如图11-34 b所示）。可以选择*A*端或*B*端进行通电，在相应线圈中产生电流，继而形成磁场吸引电机中部的转子旋转到期望的方向。图11-34 b描绘出了在定子*A*被通电的情况下，将吸引转子旋转到图中所示的方向。与此类似，横向相互串联的两个线圈（端头分别为*C*和*D*）分别缠绕在左和右两个定子铁芯上面，而且对其通电继而吸引转子旋转的原理和纵向的两个线圈是相同的。

关于转子和定子之间相互作用更详细的描述请参考图11-35，假设定子*A*最先通电变成电磁铁，将吸引转子旋转到如图11-35a所示的方向；接下来是定子*D*通电，产生电磁作用吸引转子转到如图11-35b所示的方向；然后是定子*B*，吸引转子旋转到图11-35c所示的方向；最后是定子*C*通电也产生类似的旋转效应。按照*A*、*D*、*B*、*C*的顺序分别激发各个定子线圈将使得转子按顺时针方向旋转一周，或者说转子旋转了360°。相反，如果按照与此相反的方向激发各个定子线圈，将会使转子沿相反的逆时针方向旋转。 310

如果步进电机是按上述方式运行的（即旋转一周有4个特定位置，每一步旋转90°），那么这种步进电机被称为**4拍步进电机**。如果需要更高的精度，可选择**8拍步进电机**，每一步旋转45°。在激发定子线圈的过程中，对于两个相邻定子线圈之间的位置，如果只激发一个线圈是无法达到的，需要同时激发这两个相邻的线圈才能将定子转到两

相邻线圈之间角分线的方向。例如，为使转子按顺时针方向旋转，需要按照 A、AD、D、DB、B、BC、C、CA的顺序来激发线圈，前三步的旋转状态如图11-36所示。

311

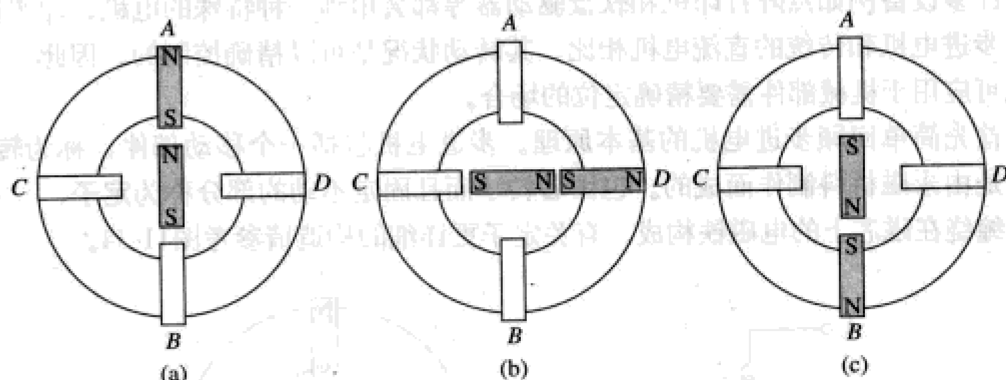


图11-35 步进电机转子旋转示意图

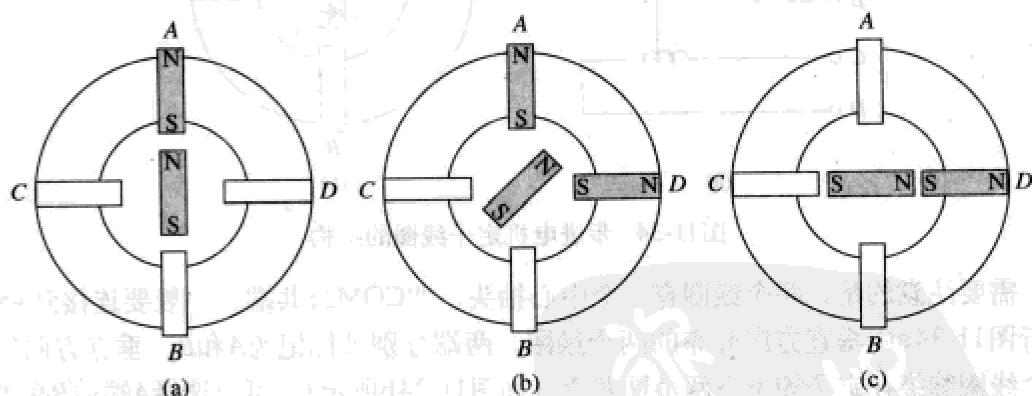


图11-36 步进电机的半拍旋转

由于8051端口的驱动电流不够，所以在和步进电机的线圈相连接时，需要通过步进电机驱动器来提高8051的驱动能力，如图11-37所示，在本例中选用的是ULN2003步进电机驱动器芯片。

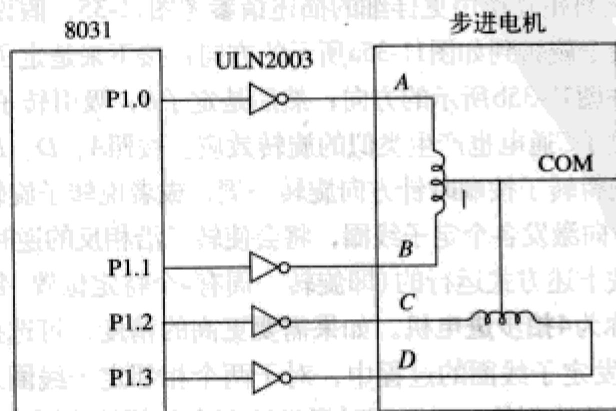


图11-37 步进电机的接口电路图

观察图11-37可知，如果要激发相应的定子线圈，只需在与其相连的8051端口引脚写1即可使电流流过线圈产生磁场效应。关于4拍步进电机和8拍步进电机的驱动逻辑时序请分别参阅表11-7和表11-8。

表11-7 4拍步进电机顺时针旋转驱动时序

拍	线圈A	线圈B	线圈C	线圈D
1	1	0	0	0
2	0	0	0	1
3	0	1	0	0
4	0	0	1	0

表11-8 8拍步进电机顺时针旋转驱动时序

拍	线圈A	线圈B	线圈C	线圈D
1	1	0	0	0
2	1	0	0	1
3	0	0	0	1
4	0	1	0	1
5	0	1	0	0
6	0	1	1	0
7	0	0	1	0
8	1	0	1	0

312

例11-14 设计目标

编写程序驱动步进电机顺时针方向旋转。如果连在8051 $\overline{INT0}$ 的开关产生一个下降沿，则改变电机的旋转方向，在本例中是逆时针方向。

程序清单如图11-38所示。外部中断0（$\overline{INT0}$）用于探测由高到低的电平跳变，一旦探测到了，程序立即转到相应的中断服务子例程（EX0ISR）执行，在中断服务子例程里将方向位D取反。两个子例程CW和CCW分别驱动步进电机按顺时针和逆时针方向旋转。在改变电机旋转方向时需要1s的延时。

```
 1      $DEBUG
 2      $NOSYMBOLS
 3      $NOPAGING
 4      ;FILE: STEPPER.SRC
 5      ;*****************************************************
 6      ;          STEPPER MOTOR INTERFACE EXAMPLE          *
 7      ;                                                   *
 8      ; This program uses the 8-step sequence to          *
 9      ; initially rotate the stepper motor clockwise.*
10      ; If a switch connected to INT0 makes a             *
11      ; high-to-low transition, the rotation is           *
12      ; changed to counterclockwise.                      *
13      ;*****************************************************
```

图11-38 步进电机接口程序

```
                     14
  0009               15     STEP1     EQU    1001B  ;8-step sequence for
  0008               16     STEP2     EQU    1000B  ; clockwise rotation
  000C               17     STEP3     EQU    1100B
  0004               18     STEP4     EQU    0100B
  0006               19     STEP5     EQU    0110B
  0002               20     STEP6     EQU    0010B
  0003               21     STEP7     EQU    0011B
  0001               22     STEP8     EQU    0001B
  0000               23     D         EQU    0      ;direction bit
                     24                             ; at bit address 0
  0064               25     HUNDRED   EQU    100    ;100 x 10000 us = 1 sec
  D8F0               26     COUNT     EQU    -10000
                     27
0000                 28               ORG 0
0000 020006          29               LJMP MAIN
                     30                             ;EXT 0 vector at 0003H
0003 B200            31     EX0ISR:   CPL D         ;complement direction bit
0005 32              32               RETI
                     33
0006 75A881          34     MAIN:     MOV IE, #81H  ;enable INT0
0009 D288            35               SETB IT0      ;negative edge triggered
000B 758901          36               MOV TMOD, #01 ;timer 0 in mode 1
000E C200            37               CLR D         ;initialize D = 0
0010 120020          38               CALL CW       ;initial: CW rotation
0013 200005          39     REPEAT:   JB D, GO_CCW  ;CCW?
0016 120020          40     GO_CW:    CALL CW       ;no: CW
0019 80F8            41               SJMP REPEAT
001B 120035          42     GO_CCW:   CALL CCW      ;yes: CCW
001E 80F3            43               SJMP REPEAT   ;repeat forever
                     44
                     45     ;*************************************************
                     46     ; Subroutine for clockwise (CW) rotation         *
                     47     ;*************************************************
                     48
0020 90005F          49     CW:       MOV DPTR, #SEQ      ;point to table
0023 E4              50               CLR A            ;point to first step
0024 C0E0            51     NEXT:     PUSH ACC         ;backup index in A
0026 93              52               MOVC A,
0027 540F            53               ANL A, #0FH      ;mask off upper 4 bits
0029 F590            54               MOV P1, A        ;send to stepper motor
002B 12004B          55               CALL DELAY       ;wait 1 second
002E D0E0            56               POP ACC          ;restore index into A
0030 04              57               INC A            ;point to next step
0031 B408F0          58               CJNE A, #8, NEXT ;ensure 8 steps
0034 22              59               RET
0046 14              73               DEC A            ;point to next step
0047 B4FFF0          74               CJNE A, #0FFH, NEXT2 ;ensure 8 steps
004A 22              75               RET
                     76
                     77     ;*************************************************
                     78     ; Subroutine for 1-second delay                  *
                     79     ;*************************************************
                     80
004B 7F64            81     DELAY:    MOV R7, #HUNDRED
004D 758CD8          82     AGAIN:    MOV TH0, #HIGH COUNT
```

图11-38 （续）

```
0050 758AF0     83            MOV TL0, #LOW COUNT
0053 D28C       84            SETB TR0
0055 308DFD     85    WAIT:   JNB TF0, $
0058 C28D       86            CLR TF0
005A C28C       87            CLR TR0
005C DFEF       88            DJNZ R7, AGAIN
005E 22         89            RET
                90
                91    ;*********************************************************
                92    ; 8-step sequence pattern for clockwise                  *
                93    ; rotation.                                               *
                94    ;*********************************************************
                95
005F 09         96    SEQ:  DB STEP1          ;lookup table with
0060 08         97          DB STEP2          ; sequence pattern
0061 0C         98          DB STEP3          ; for clockwise rotation
0062 04         99          DB STEP4
0063 06        100          DB STEP5
0064 02        101          DB STEP6
0065 03        102          DB STEP7
0066 01        103          DB STEP8
               104
               105          END
```

图11-38 （续）

主程序首先将步进电机初始化为沿顺时针方向旋转，直到一周完成后，检查方向位*D*，根据*D*的指示确定究竟是应该调用CW还是CCW子例程，来驱动步进电机连续旋转。

313～314

小结

本章讨论了很多接口方面的例子，介绍了一些采用8051进行复杂接口设计的基本概念。

但这些知识不能取代动手实验，本章以及以前各章出现的设计例子，只有在实际中经过反复的调试－纠错，才能很好地理解这些例子。如果想掌握好本章中介绍的一些概念，最好的方法是在实际的系统（如SBC-51）上亲手实践这些例子。本章为学生提供了一些将来利用微控制器（如8051）设计最小器件的基础。

习题

11.1 编程实现，利用图11-9所示的扬声器接口电路重复播放图11-39所示的乐谱。

11.2 采用8051的串行端口（模式0）作为时钟线和数据线，重新设计图11-15所示的74HC165接口电路，同时重新编写相应的程序代码。

11.3 如果采用下面的程序在图11-23所示的D/A转换电路中产生锯齿波信号（假设晶振频率为12MHz），试回答3个问题：

```
STEP   EQU     1
MAIN   MOV     P1,A
```

315

```
ADD             A,#STEP
SJMP            MAIN
```

(1) 锯齿波的频率是多少？

(2) STEP取何值时能产生大约10kHz的输出频率？

(3) 推导出由STEP求输出频率的计算公式。
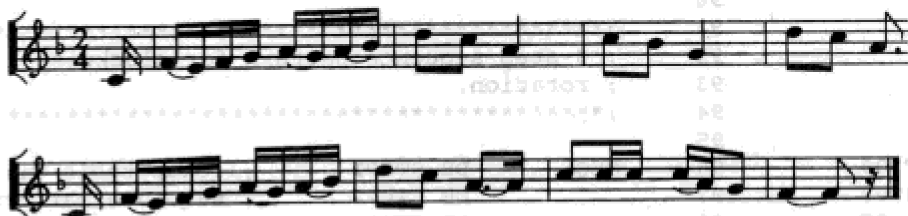


图11-39 题11.1的乐谱

11.4 本章中的几个程序都调用了MON51的子例程。如果在某程序中需要调用ISDIG子例程（检查一个字节是否为ASCII数字），符号ISDIG对应的地址是多少？

11.5 编写程序，利用8155的定时器和外部中断0在P1.7产生1kHz的方波信号（提示，请参考8155的数据文档）。

11.6 假设将图11-15所示电路中的74HC165芯片扩展到6片，即可提供48条外部输入线。试回答下列问题：

(1) 为了适用于扩展后的48条输入线，应该如何修改图11-16所示的程序？

(2) 在8051内部RAM的何处存放读入的数据？

(3) 子例程GET_BYTE的执行时间变成多少？

(4) 如果将GET_BYTE嵌入到某中断服务程序里，改中断服务程序每秒执行1次，则花费在读取48个输入信号的时间占CPU运行时间的百分比为多少？

11.7 在图11-25所示的程序清单中，利用下面的汇编指令可将常数STEP定义在内部RAM的50H存储单元：

```
STEP   DATA   50H
```

上述定义是正确的，但如果采用下面的定义也能正常工作：

```
STEP   EQU   50H
```

对于后者，ASM在汇编源程序时不会对其进行类型检查。请举出1个例子证实：如果STEP是采用EQU指令来定义的，且出现了错误，但ASM51不会给出错误提示信息，如果STEP是采用DATA指令来定义的，ASM将会给出错误提示。

11.8 假设16个键的十六进制数字键盘和8051的连接电路如图11-40所示。列号从左向右排列，第0列开始。行号从上到下排列，第0行开始。如果探测到有某个键被按下，将调用HIT子例程，寄存器R6存储列信息，P1的高4位存储行信息：

R6=3	第0列	P1.4=0	第0行
R6=2	第1列	P1.5=0	第1行
R6=1	第2列	P1.6=0	第2行
R6=0	第3列	P1.7=0	第3行

编写子例程HIT，利用R6和P1的值计算出按键所对应的十六进制数字。

316

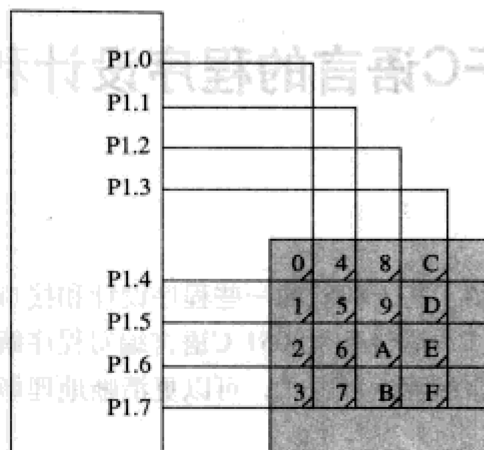


图11-40　十六进制键盘

11.9　重新编写图11-8所示电路的LCD驱动程序，使其连续显示如下信息：Welcome to the 8051 Microcontroller Experience. Hope you enjoy your reading adventure.。

11.10　前面已经讨论了8255的模式0。考虑8255其他更复杂的工作模式，并写出相应的汇编代码初始化这些模式。

11.11　比较串行接口和并行接口，你认为二者的显著区别是什么？为什么串行接口一般用于键盘、鼠标和调制解调器，而并行接口一般用于打印机？

11.12　写出图11-33所示程序清单对应的伪码。

11.13　调查步进电机的应用类型。选择其中的3种给出详细解释，例如为什么选用步进电机等。

317

第12章　基于C语言的程序设计和接口实例

12.1　引言

在前一章中，已经介绍了基于8051的一些程序设计和接口的实例，相应的程序都是采用汇编语言编写的。本章将采用8051 C语言编写程序解决同样的问题。通过对同一个项目采用两种语言编程这种方式，可以更清晰地理解两种语言的不同和相似之处。

12.2　十六进制键盘接口

在前一章中讨论了十六进制键盘接口，其设计要求是编写一个程序，使十六进制字符能通过一个4×4的键盘不断被读入，并且将相应的ASCII码传送到控制台。例12-1是一个相似的8051 C程序。

例12-1　十六进制键盘接口

采用C语言重新编写图11-4中的十六进制键盘接口的汇编语言程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

unsigned char R3, R5, R6, R7, tempA;
unsigned char bdata A;          /* represents ACC */
sbit aMSB = A ^ 7;              /* variable aMSB to refer to
                                   A.7 */
sbit aLSB = A ^ 0;              /* variable aLSB to refer to
                                   A.0 */
bit PY;                         /* represents parity bit */

void IN_HEX(void);
void HTOA(void);
void OUTCHR(void);              /* taken & modified from
                                 Example 8-6 */

bit GET_KEY(void);
void RL_A(void);
bit RRC_A(bit);
void PARITY(void);

main( )
{
while (1)                       /* repeat forever */
```

```
        {
        IN_HEX();               /* get code from keypad */
        HTOA();                 /* convert to ASCII */
        OUTCHR();               /* echo to console */
        }
}

void IN_HEX(void)
{
R3 = 50;                        /* debounce count */
while (R3 != 0)
        {
        if (GET_KEY() == 0)     /* key pressed? */
                R3 = 50;        /* no: check again */
        else
                R3--;           /* yes: repeat 50 times */
        }

tempA = A;                      /* save hex code */
R3 = 50;                        /* wait for key up */
while (R3 != 0)
        {
        if (GET_KEY() == 1)     /* key pressed? */
                R3 = 50;        /* yes: keep checking */
        else
                R3--;           /* no: repeat 50 times */
        }
A = tempA;                      /* recover hex code and return
                                   */
}

bit GET_KEY(void)
{
bit C = 0;                      /* default return value: no key
                                   press */

A = 0xFE;                               /* start with column 0 */

for (R6 = 4; R6 > 0; R6--)              /* check all 4 columns */
        {
        P1 = A;                         /* activate column line
                                           */
        A = P1;                         /* read back Port 0 */
        tempA = A&(0xF0);               /* isolate row lines */
        if (tempA == 0xF0)              /* row lines active? */
              RL_A();                   /* no: move to next
                                           column line */
        else
              {
              R7 = A;                   /* save in R6 */
              A = 4;                    /* prepare to calculate
                                           col. weighting */
              C = 0;
              A = A - R6;               /* 4 - R6 = weighting */
              R6 = A;                   /* save in R6 */
              A = R7;                   /* restore scan code */
              A = ( R7 << 4 | R7 >> 4 );/* put in low nibble */
```

```
            for (R5 = 4; R5 > 0; R5--)/* use R5 as counter */
                {
                C = RRC_A(C);          /* rotate ACC until 0 */
                if (C == 0)
                        break;         /* done when C = 0 */
                else
                        R6 = R6 + 4;   /* add 4 until row found
                                          */
                }
            C = 1;                     /* key pressed! */
            A = R6;                    /* code in ACC */
            return C;
            }
        }
return C;
}

void RL_A(void)
{
bit tempBit;

tempBit = aMSB;                        /* backup MSB of A */
A = A << 1;                            /* rotate A left */
aLSB = tempBit;                        /* rotate LSB into MSB */
}

bit RRC_A(bit C)
{
bit tempBit;

tempBit = C;                           /* backup C */
C = aLSB;                       /* rotate A.0 into C */
A = A >> 1;                     /* rotate A right */
aMSB = tempBit;                 /* rotate C into A.7 */
return C;
}

void HTOA(void)
{
A = A & 0xF;                    /* ensure upper nibble clear */
if (A >= 0xA)                   /* 'A' to 'F'? */
    A = A + 7;                  /* yes: add extra */
A = A + '0';                    /* no: convert directly */
}


void OUTCHR(void)
{
PARITY();                       /* get even parity of A and put
                                   in PY */

PY = !PY;                       /* change to odd parity */
aMSB = PY;                      /* add to character code */
while (TI != 1);                /* Tx empty? no: check again */
TI = 0;                         /* yes: clear flag and */
SBUF = A;                       /* send character */
aMSB = 0;                       /* strip off parity bit */
}
```

```
void PARITY(void)
{
int i;
PY = 0;                          /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )       /* calculate parity of A*/
        PY ^= ( A >> i ) & 1;
}
```

讨论：

前文中介绍过，寄存器R0~R7用于存储C语言函数中的参数。因此，用C语言进行8051编程时，通常不建议使用这些寄存器存储临时参数。替代方案是，定义一些8位的变量来完成该任务。在上面的例子中，故意将这些变量命名为R3~R7，目的就是方便读者建立C程序和汇编程序（图11-4）之间的关联，进而很容易理解对应的C语言程序。

基于同样的原因，由于累加器ACC在程序执行过程中要被频繁使用并经常重写入数据，所以一般不能在C程序中使用。在上例中，使用可位寻址变量*A*来代表累加器。

正因为不能直接使用ACC，而是代之以变量*A*，所以对应的奇偶标志*P*也不能根据*A*的奇偶检验性质自动更新。解决的方案是，在程序中定义一个函数来实现这个功能。因此，对OUTCHR()函数做了少许改动，即引入了一个PARITY()函数，用于计算*A*的奇偶校验标志，并保存在位变量PY里。

320~322

在该程序里还引入了两个函数，即RL_A()和RRC_A()，其功能分别和汇编语言的RL A和RRC A的功能相同。之所以这么做是因为在C语言中没有直接支持循环移位的操作或函数，只有相左或向右移位的操作指令（如<<）。

12.3 多个七段LED接口

在第11章中介绍了一个采用汇编语言编写的七段LED与8051接口的程序。该程序能将BCD数字存储到地址为70H~71H的内部RAM中，并且可以通过使用中断将这些数字发送到LED显示器上，输出频率为每秒钟10次。下面将给出用C语言编写程序的粗略大纲。允许读者根据个人的参数选择进行修改。

例12-2 七段LED接口设计

采用C语言重新编写图11-6中的七段LED接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

unsigned char bdata A;          /* represents ACC */
int xdata * idata DPTR;         /* DPTR in idata, points to
                                   xdata */
```

```
sbit DIN = P1^7;                    /* MC14499 interface lines */
sbit CLOCK = P1^6;
sbit ENABLE = P1^5;
sbit aLSB = A^0;
sbit aMSB = A^7;
int xdata X8155 = 0x100;            /* 8155 address */
int xdata TIMER = X8155 + 4;        /* timer registers */
int count = 4000;                   /* interrupts @ 2000 μs */
unsigned char mode = 0x40;          /* timer mode bits */

int digits, icount;
unsigned char tempA;

void UPDATE(void);
void OUT8(void);
bit RLC_A(bit);

main( )
{
DPTR = TIMER;                      /* initialize 8155 timer */
A = 0xA0;                          /* low byte of count for 500 Hz
                                      square wave */
DPTR[TIMER] = A;
TIMER++;
A = 0xF;                           /* high byte of count */
DPTR[TIMER] = A;
A = 0xC0;                       /* start timer command */
DPTR[X8155] = A;                /* send to 8155 command register */
icount = 50;                    /* initialize int. counter */
EA = 1;                         /* enable interrupts */
EX0 = 1;                        /* enable external 0 interrupt */
IT0 = 1;                        /* negative-edge triggered */
while(1);                       /* do nothing */
}

void EX0ISR(void) interrupt 0
{
if (--icount == 0)              /* on 50th interrupt */
        {
icount = 50;                    /* reset counter and */
UPDATE();                       /* refresh LED display */
}
}

void UPDATE(void)
{
tempA = A;                      /* save A */
ENABLE = 0;                     /* prepare MC14499 */
A = digits;                     /* get first two digits */
OUT8();                         /* send two digits */
A = digits + 1;                 /* get second byte */
OUT8();                         /* send last two digits */
ENABLE = 1;                     /* disable MC14499 */
A = tempA;                      /* restore A */
}

void OUT8(void) using 0
```

```
{
for (tempA = 8; tempA > 0; tempA--)
        {
        CY = RLC_A(CY);         /* put bit in C flag */
        DIN = CY;               /* send it to MC14499 */
        CLOCK = 0;              /* 3 µs low pulse on clock line */
        CLOCK = 1;
        }
}

bit RLC_A(bit C)
{
bit tempBit;

tempBit = C;                    /* backup C */
C = aMSB;                       /* shift A.7 into C */
A = A << 1;                     /* shift A left */
aLSB = tempBit;                 /* shift C into A.0 */
return C;
}
```

讨论：

在8051中，DPTR作为指针指向外部存储器空间。为了使C语言程序与对应的汇编程序直接联系在一起，需要预先声明int xdata * idata DPTR，这个语句说明DPTR是一个存储在idata存储区（SFR所在的区域）中的16位指针变量，且该指针指向xdata存储区。在汇编语言中，查找表中的元素是通过指令MOVX @DPTR, A或MOVC @DPTR, A获得的，并且A中的内容作为特定元素的索引值；而DPTR则保存了表的基址（开始地址）。在C语言中与这种情形很相似。可以使用DPTR指针变量来指向一个数组的第一个元素（这可认为是C语言中的查找表操作）。因此，DPTR [A]语句用来引用该数组中的一个特定元素，这里A是数组的索引值。

12.4 液晶显示器接口

11.5节给出了一个液晶显示器（LCD）的接口设计实例。该程序不断地从地址为30H~7FH的内部RAM中读取ASCII字符并显示到液晶显示器上，每次16个字符。采用C语言重写的相同程序如例12-3所示。

例12-3 LCD接口

采用C语言重新编写图11-8中的LCD接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

sbit RS = P3^0;
```

```
sbit RW = P3^1;
sbit E = P3^2;
sbit busy = P1^7;
unsigned char bdata A;   /* represents ACC */
unsigned char * ptr;
unsigned char count;
void INIT(void);
void NEW(void);
void DISP(void);
void WAIT(void);
void OUT(void);

main( )
{
INIT();                          /* initialize LCD */
count = 16;                      /* initialize character count */
while(1)
        {
        for (ptr = 0x30; ptr < 0x80; ptr++, count--)
        {
        A = *ptr;                 /* get next character */
        DISP();                   /* display on LCD */
        if (count==0)
              {
              count = 16;  /* if end of line, reinitialize count */
              NEW();       /*  and refresh LCD */
              }
        }
        }
}
void INIT(void)
{
A = 0x38;                         /* 2 lines, 5 × 7 matrix */
WAIT();                           /* wait for LCD to be free */
RS = 0;                           /* output a command */
OUT();                            /* send it out */
A = 0x0E;                         /* LCD on, cursor on */
WAIT();                           /* wait for LCD to be free */
RS = 0;                           /* output a command */
OUT();                            /* send it out */
NEW();                            /* refresh LCD display */
}
void NEW(void)
{
A = 0x01;                         /* clear LCD */
WAIT();                           /* wait for LCD to be free */
RS = 0;                           /* output a command */
OUT();                            /* send it out */

A = 0x80;                         /* cursor: line 1, position 1 */
WAIT();                           /* wait for LCD to be free */
RS = 0;                           /* output a command */
OUT();                            /* send it out */
}
```

325

```
void DISP(void)
{
WAIT();                     /* wait for LCD to be free */
RS = 1;                     /* output a data */
OUT();                      /* send it out */
}

void WAIT(void)
{
do
        {
        RS = 0;             /* command */
        RW = 1;             /* read */
        busy = 1;           /* make busy bit = input */
        E = 1;              /* 1-to-0 transition to */
        E = 0;              /* enable LCD */
        }
while (busy);               /* if busy, wait */
}

void OUT(void)
{
P1 = A;                     /* get ready output to LCD */
RW = 0;                     /* write */
E = 1;                      /* 1-to-0 transition to */
E = 0;                      /* enable LCD */
}
```

讨论：

在这个例子里，需要不断访问内部RAM空间。在汇编语言里，这是通过间接寻址完成的。在C语言中，通过指针来完成，由于指针的位置在内部RAM中，所以ptr将通过语句unsigned char data * ptr来定义。在这里使用unsigned char是因为，存储单元里存放的是8位的数据。程序的剩余部分简明易懂，不再赘述。

12.5 扬声器接口

在11.6节中给出了一个能连续播放A大调、中断驱动的汇编程序。在下面的例12-4中给出了用C语言编写的相应程序。

例12-4 扬声器接口

采用C语言重新编写图11-10中的扬声器接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

#define LENGTH 12
```

326
327

```
sbit outbit = P1^7;
int reload;                 /* use this to temporarily store the
                               reload */
                            /* value for the current note    */
int code * PC;              /* PC points to the TABLE */
int REPEAT = 5;             /* reload value = -50000 causes 0.05
                               sec per */
                            /* timeout. Do 5 times, to get 5 x
                               0.05   */
                            /*  = 0.25 seconds per note    */
int ncount, tcount;         /* note counter & timeout counter */
int i, j;
                              /* look-up table of notes in A
                                 major scale */
int code TABLE[LENGTH] = {-1136, -1136, -1012, -902, -851,
                          758, -676,
                          -602, -568, -568, -568, -568};

void GETVAL(void);

main()
{
TMOD = 0x11;                  /* both timers 16-bit mode */
ncount = 0;                   /* initialize note counter to 0 */
tcount = REPEAT;              /* initialize timeout counter to 5
                                 */

IE = 0x8A;                    /* timer 0 & 1 interrupts on */
TF1 = 1;                      /* force Timer 1 interrupt */
TF0 = 1;                      /* force Timer 0 interrupt */
while(1);                     /* ZzZzZzZz time for a nap */
}

void T0ISR(void) interrupt 1
{
TR0 = 0;                      /* stop timer */
TH0 = 0x3C;                   /* HIGH(-50000) */
TL0 = 0xB0;                   /* LOW(-50000) */
if (--tcount == 0)
      {
      tcount = REPEAT;        /* if 5th int, reset */
      ncount++;               /* increment note */
      if (ncount == LENGTH)   /* if beyond last note... */
            ncount = 0;       /* reset, A = 440Hz */
      }
TR0 = 1;                      /* start timer, go back to ZzZzZzZ
                                 */

}

void T1ISR(void) interrupt 3
{
outbit = !outbit;             /* music maestro! */
TR1 = 0;                      /* stop timer */
reload = ncount;              /* get note counter */
GETBYTE();                    /* get 2 bytes of reload value */
TH1 = reload >> 8;            /* put high byte in timer high
                                 register */
```

```
TL1 = reload & 0xFF;          /* put low byte in timer low
                                 register */
TR1 = 1;                      /* start timer */
}

void GETVAL(void)             /* table look-up function */
{
PC = TABLE;                   /* point to TABLE */
reload = PC[reload];          /* read from TABLE into reload */
}
```

讨论：

图11-10中的汇编语言解决方案是使用程序存储器中的查找表。为了在C语言编程中实现同样的目的，我们使用存储在程序存储器中的数组。在上面的答案中，这个数组被命名为TABLE。从数组中获取元素的任务是通过函数GETVAL()完成的，它通过PC指针访问数组元素。与图11-10汇编语言解决方案相比，要注意一点，在C语言程序里不必从表中连续读取两次数据了（每次读1个字节）。这是因为在C语言中，允许定义表示重载值的16位数据作为数组的1个元素，而且可以将每次的重载值作为1个元素进行读取操作，本例就是这么做的。因此，仅需从表中读取一次即可获得16位的重载值。然后，右移8位得到重载值的高字节，屏蔽掉高8位（令重载值同0XFF相与）获得其低字节。

12.6 非易失性RAM接口

11.7节说明了非易失性RAM(NVRAM)是如何保持其内容的，即使在断电的情况下也不会发生内容丢失。11.7节还给出了一个接口程序，说明如何从8051复制数据到NVRAM上和从NVRAM读回已存储的数据。在下面的例12-5中给出了对应的C语言程序。

例12-5 NVRAM接口

采用C语言重新编写图11-13中的NVRAM接口程序。

答案：

```
#include <reg51.h>
#include <reg51.h>
#include <stdio.h>

#define RECALL 0x85          /* X2444 recall instruction */
#define WRITE 0x84           /* X2444 write enable instruction */
#define STORE 0x81           /* X2444 store instruction */
#define SLEEP 0x82           /* X2444 sleep instruction */
#define W_DATA 0x83          /* X2444 write data instruction */
#define R_DATA 0x87          /* X2444 read data instruction */
#define LENGTH 32            /* 32 bytes saved/restored */

unsigned char * R0;         /* used to point to NVRAM locations */
```

328
~
329

```
unsigned char R5, R6, R7;
unsigned char bdata A;     /* represents accumulator */
sbit aMSB = A ^ 7;         /* variable aMSB to refer to A.7 */
sbit aLSB = A ^ 0;         /* variable aLSB to refer to A.0 */
sbit DIN = P1^2;           /* X2444 interface lines */
sbit ENABLE = P1^1;
sbit CLOCK = P1^0;
bit C;
unsigned char NVRAM[LENGTH] _at_ 0x60;

void SAVE(void);
void RECOVER(void);
void R_BYTE(void);
void W_BYTE(void);
void RL_A(void);
bit RLC_A(bit);

main()
{
while(1)                      /* repeat forever */
       {
       SAVE();                /* copy from 8051 internal
                                 locations 60H-7FH */
                              /*  to X2444 EPROM     */
       RECOVER();             /* read previously saved data from
                                 X2444 EPROM to */
                              /*   locations 60H-7FH      */
       }
}

void SAVE(void)
{
R0 = NVRAM;                   /* R0 to point to locations to save
                                 to */
ENABLE = 0;                   /* disable X2444 */
A = RECALL;                   /* recall instruction */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
A = WRITE;                    /* write enable prepares X2444 to
                                 be written to */
ENABLE = 1;
W_BYTE();
ENABLE = 0;
for (R7 = 0; R7 < 16; R7++)/* R7 = X2444 address */
     {
     A = R7;                  /* put address in A */
      RL_A();                 /* put in bits 3, 4, 5, 6 */
     RL_A();
     RL_A();
     A = A|W_DATA;            /* build write instruction */
     ENABLE = 1;
     W_BYTE();
     for (R5 = 2; R5 > 0; R5--)
         {
```

```
                A = *R0;            /* get 8051 data */
                R0++;               /* point to next byte */
                W_BYTE();           /* send byte to X2444 */
                }
        ENABLE = 0;
    }
    A = STORE;                      /* if finished, copy to EPROM */
    ENABLE = 1;
    W_BYTE();
    ENABLE = 0;
    A = SLEEP;                      /* put X2444 to sleep */
    ENABLE = 1;
    W_BYTE();
    ENABLE = 0;
    }

    void RECOVER(void)
    {
    R0 = NVRAM;
    ENABLE = 0;
    A = RECALL;                     /* recall instruction */
    ENABLE = 1;
    W_BYTE();
    ENABLE = 0;
    for (R7 = 0; R7 < 16; R7++)/* R7 = X2444 address */
        {
        A = R7;                     /* put address in A */
        RL_A();                     /* build read instruction */
        RL_A();
        RL_A();
        A = A|R_DATA;
        ENABLE = 1;
        W_BYTE();                   /* send read instruction */
        for (R5 = 2; R5 > 0; R5--)
            {
            R_BYTE();               /* read byte of data */
            *R0 = A;                /* put in 8051 RAM */
            R0++;                   /* point to next location */
            }
        ENABLE = 0;
    }
    A = SLEEP;                      /* put X2444 to sleep */
    ENABLE = 1;
    W_BYTE();
    ENABLE = 0;
    }

    void R_BYTE(void)
    {
    for (R6 = 8; R6 > 0; R6--) /* use R6 as bit counter */
        {
        C = DIN;                    /* put X2444 data bit in C */
        C = RLC_A(C);               /* build byte in Accumulator */
        CLOCK = 1;                  /* toggle clock line (1 us) */
        CLOCK = 0;
```

```
        }
    }

    void W_BYTE(void)
    {
    for (R6 = 8; R6 > 0; R6--) /* use R6 as bit counter */
        {
        C = RLC_A(C);          /* put bit to write in C */
        DIN = C;               /* put in X2444 DATA IN line */
        CLOCK = 1;             /* clock bit into X2444 */
        CLOCK = 0;
        }
    }

    void RL_A(void)
    {
    bit tempBit;

    tempBit = aMSB;            /* backup MSB of A */
    A = A << 1;                /* rotate A left */
    aLSB = tempBit;            /* rotate LSB into MSB */
    }

    bit RLC_A(bit C)
    {
    bit tempBit;

    tempBit = C;               /* backup C */
    C = aMSB;                  /* shift ACC.7 into C */
    A = A << 1;                /* shift ACC left */
    aLSB = tempBit;            /* shift C into ACC.0 */
    return C;
    }
```

讨论：

这个程序将32字节数据分配到内部RAM中，地址为60H~70H。这可以通过定义一个包含32个元素（每个元素为1个字节）的数组NVRAM，并指定其绝对地址60H的方式来实现。主程序不断地执行：将内部RAM的60H~70H单元的数据存储到NVRAM中，再将原先存储在NVRAM的数据读取出来，重新存储到内部RAM的60H~70H单元。从NVRAM数组中读取元素是通过一个指针变量R0间接完成的。数据不断地通过函数W_BYTE()和R_BYTE()来分别完成写入和读出数据到NVRAM。

330~332

12.7 输入/输出扩展

在上一章中介绍了两种基于汇编语言的输入/输出（I/O）扩展方法。下面的两个例子说明了如何采用C语言实现相同的I/O扩展。例12-6详细地介绍了通过使用移位寄存器实现I/O扩展的方法，而例12-7则说明了如何采用8255实现I/O扩展。

例12-6 移位寄存器接口

采用C语言重新编写图11-16中的移位寄存器接口程序。

答案：

```
#include <reg51.h>
#define COUNT 2                             /* number of shift registers */
unsigned char bdata * R0;
unsigned char R6, R7;
unsigned char xdata * idata DPTR;           /* DPTR in data, points to xdata */
unsigned char bdata A;                      /* represents ACC */
sbit aMSB = A^7;
sbit aLSB = A^0;
sbit SHIFT = P1^7;                          /* SHIFT/LOAD input: 1 = shift,
                                               0 = load */
sbit CLOCK = P1^6;                          /* CLOCK input */
sbit DOUT = P1^5;                           /* DATA OUT output */
bit C;
bit PY;                                     /* represents PARITY bit */
char * BANNER = {"*** TEST 74HC165 INTERFACE ***\n"};
unsigned char bdata BUFFER[COUNT]; /* buffer to store bytes read */
void GET_BYTES(void);
void SEND_HELLO_MESSAGE(void);
void DISPLAY_RESULTS(void);
bit RRC_A(bit C);
void OUTSTR(void);
void OUTCHR(void);
void OUT2HEX(void);
void SWAP_A(void);
void PARITY(void);
void HTOA(void);
main()
{
CLOCK = 1;                                  /* set interface lines initially in … */
SHIFT = 1;                                  /*  … case not already */
DOUT = 1;                                   /* DOUT must be set (input ) */
SEND_HELLO_MESSAGE();                       /* banner message */
while(1)                                    /* loop forever */
        {
        GET_BYTES();                        /* read shift registers */
        DISPLAY_RESULTS();            /* show results */
        }
}

void GET_BYTES(void)
{
for (R6 = COUNT; R6 > 0; R6--)        /* use R6 as byte counter */
      {
      R0 = 0x25;                      /* use R0 as pointer to buffer in bdata */
      SHIFT = 0;                      /* load into shift registers by pulsing… */
      SHIFT = 1;                      /* … SHIFT/LOAD low */
      for (R7 = 8; R7 > 0; R7--)/* use R7 as bit counter */
            {
            C = DOUT;                 /* get a bit (put it in C) */
            C = RRC_A(C);             /* put in A.0 (LSB 1st) */
            CLOCK = 0;                /* pulse CLOCK line (shifts bits towards … */
            CLOCK = 1;                /* … DATA OUT */
```

```
            }
        *R0 = A;                        /* if 8th shift, put in buffer */
        R0++;                           /* increment pointer to buffer */
        }
}
void SEND_HELLO_MESSAGE(void)
{
DPTR = BANNER;                          /* point to hello message */
OUTSTR();                               /* send it to console */
}

void DISPLAY_RESULTS(void)
{
R0 = 0x25;                              /* R0 points to bytes */
for (R6 = COUNT; R6 > 0; R6--)          /* use R6 as byte counter */
        {
        A = *R0;                        /* get byte */
        R0++;                           /* increment pointer */
        OUT2HEX();                      /* output as 2 hex character */
        A = ' ';                        /* separate bytes */
        OUTCHR();
        }                               /* repeat for each byte */
A = '\n';                               /* begin a new line */
OUTCHR();
}

bit RRC_A(bit C)
{
bit tempBit;

tempBit = C;                        /* backup C */
C = aLSB;                           /* rotate A.0 into C */
A = A >> 1;                         /* rotate A right */
aMSB = tempBit;                     /* rotate C into A.7 */
return C;
}

void OUTSTR(void)
{
while (1)
        {
        A = 0;
        A = DPTR[A];                /* get ASCII code */
        if (A == 0)                 /* if last code, done */
                break;
        OUTCHR();                   /* if not last code, send it */
        A++;                        /* point to next code */
        }
}

void OUTCHR(void)
{
PARITY();                           /* get even parity of A and put in PY */
PY = !PY;                           /* change to odd parity */
aMSB = PY;                          /* add to character code */
while (TI != 1);                    /* Tx empty? no: check again */
TI = 0;                             /* yes: clear flag and */
SBUF = A;                           /* send character */
aMSB = 0;                           /* strip off parity bit */
```

```
}

void OUT2HEX(void)
{
unsigned char tempA = A;    /* save A in tempA */
SWAP_A();                   /* send high nibble first */
A = A & 0xf;                /* mask off unwanted nibble */
HTOA();                     /* convert hex nibble to ASCII */
OUTCHR();                   /* send to serial port */
A = tempA;                  /* restore A and send low nibble */
A = A & 0xf;
HTOA();
OUTCHR();
A = tempA;
}

void SWAP_A(void)
{
A = (A >> 4) | (A << 4);    /* swap upper and lower nibbles of A */
}
void PARITY(void)
{
int i;
PY = 0;                     /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )  /* calculate parity of A*/
      PY ^= ( A >> i ) & 1;
}
void HTOA(void)
{
A = A & 0xF;                /* ensure upper nibble clear */
if ( A >= 0xA )             /* 'A' to 'F'? */
     A = A + 7;             /* yes: add extra */
A = A + '0';                /* no: convert directly */
}
```

讨论：

程序首先发送一个标题信息到与串行端口相连的显示设备。实际上是指向该信息并且调用OUTSTR()函数。该函数依次调用OUTCHR()函数，每次发送信息中的一个字符。调用的函数GETBYTES()则从移位寄存器中读取两个字节。这些字符通过调用DISPLAY_RESULTS()函数直接显示出来。GETBYTES()函数在字节数据中一位接一位地移动。同时，在调用OUTCHR()函数发送ASCII码到显示器之前，DISPLAY_RESULTS()通过使用OUT2HEX()函数将字符从十六进制转换为ASCII码形式。

11.8.2节中的设计例题讨论了如何使用8255增加三个I/O端口。给出的汇编程序可以读取与端口*A*相连的8个开关状态，对于每个闭合的开关，对应的与端口*B*相连的LED中有一个被点亮。完成此任务的C语言程序如下所示。

例12-7 8255接口

采用C语言重新编写图11-18中的8255接口程序。

333
~
336

答案：

```
#include <reg51.h>
#include <stdio.h>
unsigned char xdata * idata DPTR; /* DPTR in data, points to
                                     xdata */
unsigned char bdata A;            /* represents ACC */
sbit a_0 = A^0;
sbit a_1 = A^1;
sbit a_2 = A^2;
sbit a_3 = A^3;
sbit a_4 = A^4;
sbit a_5 = A^5;
sbit a_6 = A^6;
sbit a_7 = A^7;
void CPL_A();
main()
{
A = 0x90;                    /* Port A = input, Port B = output */
DPTR = 0x0103;               /* point to control register */
*DPTR = A;                   /* send control word */
while(1)
      {
      DPTR = 0x0100;         /* point to Port A */
      A = *DPTR;             /* read switches */
      CPL_A();               /* complement */
      DPTR = 0x0101;         /* point to Port B */
      *DPTR = A;             /* light up LEDs */
      }
}
void CPL_A(void)
{
a_0 = (!a_0);
a_1 = (!a_1);
a_2 = (!a_2);
a_3 = (!a_3);
a_4 = (!a_4);
a_5 = (!a_5);
a_6 = (!a_6);
a_7 = (!a_7);
}
```

讨论：

这个例子仍然使用指针DPTR访问外部存储器空间。通过函数CPL_A()将A的每1位取反从而得到A的反码。

12.8 RS232（EIA-232）串行接口

在11.9节中讨论了8051通过RS232 串行接口连接到PC机的程序设计。这个设计例子是编写一个程序，实现从PC机输入十进制数字，然后发送相应的ASCII码显示在PC显示器上。用C语言编写的相应程序见例12-8。

例12-8 R232接口

采用C语言重新编写图11-20中的RS232接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>
unsigned char bdata A;                  /* represents ACC */
sbit aMSB = A ^ 7;                      /* variable aMSB to refer to A.7 */
sbit RTS = P1^7;                        /* variable RTS to refer to P1.7 */
sbit CTS = P1^6;                        /* variable CTS to refer to P1.6 */
bit C;                                  /* represents carry bit */
unsigned char code * idata DPTR;        /* DPTR in data, points to code */
unsigned char code ASC[10] ={0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
   0x38, 0x39};                         /* ASCII table */
char code * MSG
 = {"PLEASE ENTER NUMBER = "};          /* initial message */
void FACE(void);
void INCHAR(void);                      /* taken & modified from Example 8-7 */
void OUTCHR(void);                      /* taken & modified from Example 8-6 */
void PARITY(void);
main()
{
FACE();                                 /* initialize serial port & do handshake */
DPTR = ASC;                             /* point to ASCII table */
while(1)
        {
        INCHAR();                       /* get decimal number and put in A */
        A = DPTR[A];                    /* convert to ASCII and store back in A */
        OUTCHR();                       /* output ASCII to serial port */
        }
}
void FACE(void)
{
TMOD = 0x20;                            /* timer 1, mode 2 */
TH1 = 0x98;                             /* reload count for 9600 baud */
SCON = 0x52;                            /* serial port, mode 1 */
DPTR = MSG;                             /* pointer to initial message */
TR1 = 1;                                /* start timer 1 */
RTS = 0;                                /* assert RTS */
while (CTS == 1);                       /* wait for CTS */
do
        {
        A = DPTR[A];                    /* get ASCII characters */
        OUTCHR();                       /* send out */
        DPTR++;                         /* if not end of message, get next character */
        }
while (A != 0);                         /* else if end of message, stop */
}

void INCHAR(void)
{
```

```
while (RI != 1);                /* wait for character */
RI = 0;                         /* clear flag */
A = SBUF;                       /* read char into A */
PARITY();                       /* get even parity of A and put in PY */
C = PY;                         /* for odd parity in accumulator, PY should
                                   be set */
C = !C;                         /* complementing correctly indicates if
                                   "error" */
aMSB = 0;                       /* strip off parity */
}

void OUTCHR(void)
{
PARITY();                       /* get even parity of A and put in PY */
PY = !PY;                       /* change to odd parity */
aMSB = PY;                      /* add to character code */
while (TI != 1);                /* Tx empty? no: check again */
TI = 0;                         /* yes: clear flag and */
SBUF = A;                       /* send character */
aMSB = 0;                       /* strip off parity bit */
}

void PARITY(void)
{
int i;
PY = 0;                         /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ )      /* calculate parity of A*/
        PY ^= ( A >> i ) & 1;
}
```

讨论：

这个例子使用了一些熟悉的概念，如使用可寻址位变量A表示累加器，在程序存储器中使用一个数组表示查找表，采用指针DPTR访问查找表。此外，在汇编语言里，允许通过指令DB或DW在程序存储器的某个位置开始连续地存储数据，这在C语言中是通过使用定义在程序存储器中的数组实现的。在例程中也用到了函数INCHAR()，这与第8章中讨论的INCHR()相似。但是由于将A变量作为累加器使用，所以我们需要通过调用函数PARITY()自行计算奇偶性(奇偶检验的结果被存储在PY中)。

12.9 CENTRONICS并行接口

打印机与计算机通过并行端口进行通信，而且所使用的基本标准被称为CENTRONICS并行接口。在上一章中给出了一个检查打印机状态、并向打印机连续发送检测信息令其打印的汇编程序。例12-9说明了该程序如何通过C语言来实现。

338
|
339

例12-9 中心并行端口的接口

采用C语言重新编写图11-22中的并行接口出程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

unsigned char bdata A;                      /* represents ACC */
unsigned char MASK = 0x3C;                  /* only check 4 of the bits */
unsigned char OK = 0x30;                    /* normal values of the 4 status bits */
sbit STR = P3^0;
sbit ACK = P3^1;
sbit BUSY = P3^2;
char code * MSG =
   {"THIS IS A TEST FOR THE PRINTER"};      /* test msg */
unsigned char code * idata DPTR;            /* DPTR in data, points to code */
main()
{
while(1)
{
      DPTR = MSG;                           /* point to test message */
      do
       {
       P3 = MASK;                           /* activate STROBE, P3.1-P3.5 as input */
       A = P3;                              /* read printer status */
       A = A & MASK;                        /* only P3.1-P3.5 are wanted */
       if (A != OK)                         /* any error? */
             return;                        /* yes: stop */
       A = 0;                               /* no error, get ready to send */
       while (ACK == 1);                    /*  before send, wait for ACK */
       A = DPTR[                            /* get char in test message */
       P1 = A;                              /* send char to printer */
       DPTR++;                              /* if not end, get next character */
       }
       while ( A != 0 );                    /* else if end of message, stop */
       }
}
```

讨论：

这里，8位变量MASK用来存储屏蔽值，使一些特定的位通过清零（和0逻辑与）方法实现屏蔽（忽略）。需要记住的是，逻辑AND运算由单个“&”符号而非由两个“&”表示。两个“&”并列而成的“&&”符号表示一个关系运算的AND。

340

12.10 模拟输出

模拟设备接口经常要用到ADC和DAC。在11.11节中介绍了如何将8051与MC1408L8 DAC芯片连接，从而通过使用查表方式产生一个模拟正弦波信号。这里介绍相应的C语言程序。

例12-10 模拟输出

采用C语言重新编写图11-25种的DAC接口程序。

答案：

```
#include <reg51.h>
#define MAX 1024
```

```
                                        /* truncated TABLE of 1024 entries */
unsigned char TABLE[MAX] = {127, 128, 129, 130, 131, ...};

int data STEP = 1;          /* can be initialized to any increment value */
unsigned char A;            /* represents ACC */
int index = 0;              /* use to point to entries in TABLE */

main()
{
TMOD = 0x2;                 /* 8-bit, auto reload */
TH0 = -100;                 /* 100 ms delay */
TR0 = 1;                    /* start timer */
IE = 0x82;                  /* enable timer 0 interrupt */
while(1);                   /* main loop does nothing! */
}

void T0ISR (void) interrupt 1
{
index = index + STEP;       /* add STEP to index */
if ( index > MAX )
      index = 0;            /* if end of TABLE, back to beginning */
A = TABLE[index];           /* get entry */
P1 = A;                     /* send it */
}
```

讨论：

主程序初始化定时器0，启用定时器0中断，然后就进入什么也不做的无限循环状态。实际上，完成任务的程序主体包含在定时器0的中断服务程序中，每次中断发生时，从TABLE[]读取数据。TABLE[]是一个含有1024个元素的数组，其数值在0~255之间，并与某正弦波信号（1个周期）的幅值相对应（注意在上述程序中，TABLE[]是以截取形式出现的）。当读到数据表最后一个元素时，数据指针被重新置0，这样下一个数据就可以再次从数组的第1个元素开始。在运行这个程序前，一个与图11-24相似的程序被用于生成TABLE[]中的1024个数据。

341

12.11 模拟输入

图11-27展示了一个汇编程序如何通过ADC读取可调电位器中心抽头输出的电压并转换成数字格式，然后再将对应的ASCII码输出到控制台。这里给出相对应的C语言程序。

例12-11 模拟输入

采用C语言重新编写图11-27中的ADC接口程序。

答案：

```
#include <reg51.h>
#define PORTA 0x101               /* 8155 Port A */

int xdata * idata DPTR;           /* DPTR in idata, points to xdata */
unsigned char bdata A;            /* represents ACC */
sbit aMSB = A ^ 7;                /* variable aMSB to refer to A.7 */
sbit aLSB = A ^ 0;                /* variable aLSB to refer to A.0 */
```

```
sbit WRITE = P1^0;          /* ADC0804 WR line */
sbit INTR = P1^1;           /* ADC0804 INTR line */
bit PY;                     /* represents parity bit */
char * BANNER = {"*** TEST ADC0804 ***\n"};

void OUTSTR(void);
void OUTCHR(void);
void OUT2HEX(void);
void SWAP_A(void);
void PARITY(void);
void HTOA(void);

main()
{
DPTR = BANNER;              /* send message */
OUTSTR();
while(1)
    {
    WRITE = 0;              /* toggle WR line */
    WRITE = 1;
    while (INTR == 1);      /* wait for INTR = 0 */
    DPTR = PORTA;           /* init DPTR -> Port A */
    A = *DPTR;              /* read ADC0804 data */
    OUT2HEX();              /* send data to console */
    }
}
void OUTSTR(void)
{
while (1)
    {
    A = 0;
    A = DPTR[A];            /* get ASCII code */
    if (A == 0)             /* if last code, done */
        break;
    OUTCHR();               /* if not last code, send it */
    A++;                    /* point to next code */
    }
}
void OUTCHR(void)
{
PARITY();                   /* get even parity of A and put in
                               PY */
PY = !PY;                   /* change to odd parity */
aMSB = PY;                  /* add to character code */
while (TI != 1);            /* Tx empty? no: check again */
TI = 0;                     /* yes: clear flag and */
SBUF = A;                   /* send character */
aMSB = 0;                   /* strip off parity bit */
}

void OUT2HEX(void)
{
unsigned char tempA = A;    /* save A in tempA */
SWAP_A();                   /* send high nibble first */
A = A & 0xf;                /* mask off unwanted nibble */
HTOA();                     /* convert hex nibble to ASCII */
OUTCHR();                   /* send to serial port */
A = tempA;                  /* restore A and send low nibble */
A = A & 0xf;
HTOA();
OUTCHR();
A = tempA;
}
```

342

```
void SWAP_A(void)
{
A = (A >> 4) | (A << 4);    /* swap upper and lower nibbles of
                               A */
}

void PARITY(void)
{
int i;
PY = 0;                        /* initialize parity to 0 */
for ( i = 0; i < 8 ; i++ ) /* calculate parity of A*/
      PY ^= ( A >> i ) & 1;
}
void HTOA(void)
{
A = A & 0xF;                   /* ensure upper nibble clear */
if ( A >= 0xA )                /* 'A' to 'F'? */
      A = A + 7;               /* yes: add extra */
A = A + '0';                   /* no: convert directly */
}
```

讨论：

主程序首先显示了一条信息到控制台，然后触发WR线开始进行模数转换。接着等待ADC的INTR线变成低电平，标志着转换已经完成，此时8051可以通过8155端口*A*读取ADC中的数据了。这个数据在发送到控制台显示前需要将其转换成ASCII码格式。

12.12 传感器接口

这一部分，我们再次讨论8051如何与典型的温度传感器相互通信。第11章曾经给出了一个利用DS1620温度传感器来监控室温的汇编程序。当室温高于23℃时，关闭锅炉加热装置，并且发出报警声。如果温度低于17℃，打开锅炉加热装置，并且停止报警。例12-12中给出了相应的C语言程序，可用于完成相同的任务。

例12-12 DS1620接口

采用C语言重新编写图11-29中的传感器接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>
unsigned char bdata A;    /* represents ACC */
sbit aMSB = A ^ 7;         /* variable aMSB to refer to A.7 */
sbit aLSB = A ^ 0;         /* variable aLSB to refer to A.0 */
sbit DQ = P1^0;
sbit CLK = P1^1;
sbit RST = P1^2;
sbit THI = P1^3;
sbit TLO = P1^4;
sbit TCOM = P1^5;
sbit FURN = P1^6;
```

343
~
344

```
int i;
void SEND(void);
bit RRC_A(bit);
main()
{
FURN = 0;                    /* turn furnace off */
RST = 1;                     /* initiate transfer */
A = 0xC;                     /* write config */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 1;                       /* write TH */
SEND();                      /* send to DS1620 */
A = 44;                      /* TH = 44 × 0.5 °C = 22 °C */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 2;                       /* write TL */
SEND();                      /* send to DS1620 */
A = 36;                      /* TL = 36 × 0.5 °C = 18 °C */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
RST = 1;                     /* initiate transfer */
A = 0xEE;                    /* start temperature sensing */
SEND();                      /* send to DS1620 */
RST = 0;                     /* stop transfer */
while(1)                     /* keep sensing forever */
      {
      if (THI == 1)          /* if T ≥ 22 °C, furnace = off */
            FURN = 0;
      if (TLO == 1)          /* if T ≤ 18 °C, furnace = on */
            FURN = 1;
      }
}
void SEND(void)
{
for (i = 8; i > 0; i--)      /* loop for all 8 bits */
      {
      CLK = 0;               /* start clock cycle */
      CY = RRC_A(CY);        /* rotate A into CY, LSB first */
      DQ = CY;               /* send out bit to DQ */
      CLK = 1;               /* complete the clock cycle */
      }
}
bit RRC_A(bit C)
{
bit tempBit;
tempBit = C;                 /* backup C */
C = aLSB;                    /* rotate A.0 into C */
A = A >> 1;                  /* rotate A right */
aMSB = tempBit;              /* rotate C into A.7 */
return C;
}
```

讨论：

该程序的主要功能是通过函数send()来完成的，将8位数据以串行的方式（LSB在先）发送到DS1620温度传感芯片的DQ（数据输入）引脚。

12.13 继电器接口

在上一章介绍了一个非常有趣的继电器的应用例子，即人行横道交通灯系统。由8051控制的继电器有两个切换状态，一是为了让行人安全穿越马路，同时点亮车行红灯和人行绿灯；二是点亮车行绿灯和人行红灯，禁止行人横穿马路。例12-13是实现交通灯控制系统的C语言程序。

例12-13 继电器接口：人行横道交通灯系统

采用C语言重新编写图11-33的继电器接口程序。

答案：

345
346

```
#include <reg51.h>
#include <stdio.h>

sbit LEDs = P1^0;
int thousand = 100;     /* 1000 × 10000 μs = 10 secs */
int count;              /* to store the count values */

void delay(void);

main()
{
IE = 0x81;              /* enable INT0 */
IT0 = 1;                /* negative edge triggered */
TMOD = 1;               /* timer 0 in mode 1 */
LEDs = 0;               /* initially, traffic = GREEN,
                           pedestrian = RED */
while (1);              /* wait forever */
}

void delay(void)        /* 10-second delay */
for (count = thousand; count > 0; count —)
        {
        TH0 = 0x6C;
        TL0 = 0x70;
        TR0 = 1;
        while(TF0 != 1);
        TF0 = 0;
        TR0 = 0;
        }
}

void EX0ISR(void) interrupt 0
{
LEDs = 1;               /* traffic light = RED, pedestrian =
                           GREEN */
```

```
    delay();                /* wait 10 seconds */
    LEDs = 0;               /* traffic light = GREEN, pedestrian =
                               RED */
}
```

讨论：

比较本例程和上一章对应的汇编程序可以发现：二者是非常类似的。唯一差别在于函数delay()所延时的时间长度。该函数的作用是产生10s的延时允许行人横穿马路，然后再切换到交通灯的初始状态。但是，实际上delay()函数所产生的时间延迟的确切值取决于所选取的8051C编译器。为了精确地测算出该函数的时延值，可以查看由C语言函数delay()编译而成的汇编代码，而每条汇编指令的执行时间是已知的，从而可以推算出确切的时延值。对于Keil公司的μVision2 IDE，可以通过在View菜单里选择（Disassembly Window ）的方法来查看由C程序编译而成的汇编代码。

12.14 步进电机接口

步进电机主要应用在需要将元件精密定位的场合。回顾第11章的步进电机接口程序：可以驱动电机沿顺时针方向旋转，而且在旋转过程中，如果随时通过开关在8051的外部中断0（INT0）产生一个下降沿触发信号，步进电机的转动方向将颠倒过来，即开始沿逆时针方向旋转。本节将给出等效的C语言程序。

347

例12-14 步进电机接口

用C语言重新编写图11-38所示的步进电机接口程序。

答案：

```
#include <reg51.h>
#include <stdio.h>

unsigned char A;                    /* ACC mirror */
unsigned char tempA;                /* temp variable */
unsigned char code * idata DPTR;    /* DPTR in data, points to
                                       code  */
bit D;                              /* direction bit for current
                                       direction */
int HUNDRED = 100;                  /* 100 × 10000 μs = 1 sec */
                                    /* 8-step sequence for
                                       clockwise rotation */
unsigned char code SEQ[8] = {0x9, 0x8, 0xC, 0x4, 0x6, 0x2,
0x3, 0x1};

void CW(void);
void CCW(void);
void delay(void);

main()
```

```
{
IE = 0x81;                          /* enable INT0 */
IT0 = 1;                            /* negative edge triggered */
TMOD = 1;                           /* timer 0 in mode 1 */
D = 0;                              /* initialize D = 0 for
                                       clockwise rotation */

CW();                               /* initial rotation is
                                       clockwise */
while(1)                            /* repeat forever */
    {
    if (D != 1)                     /* previously clockwise? */
        CCW();                      /* no: change to clockwise */
    else
        CW();                       /* yes: change to
                                       counterclockwise */
    }
}
void CW(void)                       /* function for clockwise
                                       rotation */
{
DPTR = SEQ;                         /* point to start of table */
for (A = 0; A < 8; A++)
    {
    tempA = A;                      /* backup index in A */
    A = DPTR[A];               /* get step pattern into 4 LSBs of
                                  A */
    A = A | (P1&0xF0);         /* retain 4 MSBs of P1 and put into
                                  A */
    P1 = A;                    /* send to stepper motor */
    delay();                   /* wait for 1 sec */
    A = tempA;                 /* restore index into A */
    }
}
void CCW(void)                 /*function for counterclockwise
                                 rotation */
{
DPTR = SEQ;                    /* point to start of table */
for (A = 7; A >= 0; A—)
    {
    tempA = A;                 /* backup index in A */
    A = DPTR[A];               /* get step pattern into 4 LSBs of
                                  A */
    A = A | (P1&0xF0);         /* retain 4 MSBs of P1 and put into
                                  A */
    P1 = A;                    /* send to stepper motor */
    delay();                   /* wait for 1 sec */
    A = tempA;                 /* restore index into A */
}
}

void delay(void)               /* 1 second delay */
{
for (tempA = HUNDRED; tempA > 0; tempA--)
    {
```

```
        TH0 = 0x6C;
        TL0 = 0x70;
        TR0 = 1;
        while(TF0 != 1);
        TF0 = 0;
        TR0 = 0;
        }
}

void EX0ISR(void) interrupt 0
{
D = !D;                          /* complement direction bit */
}
```

讨论：

该程序的主要函数是CW()和CCW()，二者的功能分别是驱动步进电机顺时针和逆时针旋转。两个函数的主要区别在于读取时序数据及向步进电机写数据的顺序正好相反。在向步进电机连续写两个时序数据之间要调用delay()函数，给步进电机足够的响应时间。

348
349

习题

12.1 用C语言重新编写程序，使LCD连续显示“Welcome to the 8051 Microcontroller Experience. Hope you enjoy your reading adventure.”。

12.2 假设步进电机应用于保险箱自动门系统中。步进电机将按照用户输入的一组号码来控制旋钮转动的步数，并且顺时针、逆时针、再顺时针如此交替进行。例如，输入的一组号码是：8，0，1，6，0，9，9。在这种情况下，步进电机先顺时针旋转8步，再逆时针旋转0步，然后又顺时针旋转1步……依此规律进行。

编写函数safe(int *seq, int seqsize)，接收一组有序号码而后存储在seqsize数组里。然后控制步进电机按照输入的有序号码运转。假设顺时针和逆时针旋转采用的函数分别是CW()和CCW()。

12.3 智能卡不仅可用存储器的方式构建，还可以通过微控制器来设计。微控制器可充当智能卡的大脑，这就赋予了其执行程序的能力，因此8051微控制器经常用在智能卡领域。智能卡里的某些信息是保密的，因此需要将其设置为保护格式，防止未授权者的访问。实现此功能的技术被称为信息加密。加密是将保密信息（通常称为原文）转变成不能按普通方式理解的形式（称为密文）。一旦加密完成，甚至某些人截获或通过间谍得到了这些信息，但也无法理解其真正含义。获得原始信息的唯一途径是执行和加密过程相反的操作，即解密。凯撒密码就是一种加密技术，是由其发明者Julius Caesar命名的。尽管照今天的标准看来，这种加密方法过于简单，并且已经不再用于保护信息了。这里只是希望通过凯撒密码让读者对加密技术有一个基本的认识。更多的安全处理细节请参阅第13章。

考虑英文字母表：

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

假设将所有字母左移3位，将得到新的字母表：

D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

然后，对于给出的任何需要加密的内容，就可以将其中的标准字母表中的字母用第2张字母表对应位置的字母代替，实现加密。例如，所有的A都代换成D，所有的B都代换成E，所有的C都代换成F，依此类推。

所以如下信息：

MEET ME TONIGHT

被加密成：

350

PHHW PH WRQLJKW

编写函数CaesarEncrypt(char *plain, char *cipher)，用于处理两个字符串（字符数组）plain和cipher，即加密字符串plain，并将完成加密的新字符串存储到cipher。

12.4 参考上一个问题，编写函数CaesarDecrypt(char *plain, char *cipher)解密存储在字符串cipher的已加密信息，并将解密结果存储到字符串plain里。

351

第13章 学生项目实例

13.1 引言

学生们非常热衷参与关于微控制器或其他与自动化相关的课题。当给学生分配基于8051微控制器的课程设计任务时，他们会表现出令人惊奇的热情。在本书的第11~12章中已经介绍了一些相关的设计和接口问题。作为它的扩展，这一章将给出一些学生练习设计项目。这些项目将会帮助读者把本书前面章节中所讨论的所有概念都连接成一个整体，从而加深了对8051应用系统的理解。

首先从一个最简单的项目开始讲起：基于8051的家庭安全系统。之后会有一些难度稍大一些的项目，具体包括：简单的电梯系统；井字游戏系统；基于8051的计算器；微型老鼠；会踢足球的机器人和智能卡应用系统。

13.2 家庭安全系统

基于8051的家庭安全系统是最简单的项目了，通过它可以锻炼学生们的设计技巧。它基本上就是关于如何利用8051的I/O端口与外设进行通信的一个演示。

13.2.1 项目描述

关于这样的一个家庭安全系统可以描述为：

设计一个基于8051微控制器（拥有64KB外部程序存储器和64KB外部数据存储器）的家庭安全系统。假设只有8KB的EPROM和8KB的RAM可以使用。

房屋的任何入口被打开或者有盗贼闯入，都应该能够被该安全系统侦测到：

- ❑ 大门
- ❑ 前门
- ❑ 后门
- ❑ 所有窗户

当上述任何一个入口被打开时，拉响警报并同时用七段数码管显示相应的位置号码。例如，如果大门被打开，显示1；如果某个窗户被打开，显示4。

13.2.2 系统规格

对于学生们来说，拿到这个题目的第一件事就是要确定所需要的元器件。根据

该项目的描述，可以非常清楚推断出需要下列器件：

- 1个8051微控制器
- 8片8KB ROM/EPROM/EEPROM
- 8片8KB RAM
- 4个传感器
- 1个警报器
- 1个七段数码管

系统需要64KB的外部程序存储器，而现在只有8KB ROM芯片可用，因此，只能采用8个芯片相级联的方式来构建程序存储器。外部数据存储器亦采用相同的方法处理。在项目描述中提到需要在4个不同的地点侦测可能的闯入，因此至少需要4个传感器，具体可以采用红外线、光学及开关类型的传感器。除此之外，本课题还需要采用一个报警器和一个七段数码管来指示闯入者的位置。请注意这份器材清单并不包括外部晶振和复位电路需要的元器件。关于这方面的内容，请参考第2章的图2-2。

13.2.3 系统设计

接下来的一步是设计系统的构建方案，满足之前提出的要求和规格。这个步骤包括：决定哪个器件需要接到哪里，如何利用8051的I/O端口。最后完成一张总体电路原理设计图，描述8051微控制器以及它的引脚（I/O口、控制和时钟）是如何与外部存储器和不同的I/O设备（传感器、报警器、七段数码管）连接的。学生们也很可能需要用到74LS47 BCD-七段解码器来减少所需要的I/O端口。

13.2.4 软件设计

完成硬件连接之后，学生们就可以着手设计编写软件程序来控制8051与外部设备的交互作用了。这步通常需要参考整体的电路原理图来进行。设计中会有一些需要学生自己解决的问题，对应着程序编写中你必须做出的一些决定。

- 如何探测闯入点，采用查询方式还是中断方式？
- 各个闯入点的优先级应该怎么分配？
- 当闯入事件发生时，系统需要做什么？
- 系统是不是需要具有同时探测一种以上入侵事件的能力？
- 系统是否应该具备复位和关闭的功能？
- 是否允许用户自己定制系统？
- 系统是否拥有安全性措施？例如，如何确保只有房屋主人可以定制及关闭系统？

通常，为了应对上面的这些问题，需要设计人员把自己放在使用者的角度，这

样才能更好地理解他们的处境和系统设计中应该努力解决的问题。

下面给出了家庭安全系统的一段伪代码程序：

```
WHILE [1] DO BEGIN
      WHILE [sensor == false] DO
            [wait]
      CASE [sensor] OF
            'gate' : [display = 0]
            'frontdoor' : [display = 1]
            'backdoor' : [display = 2]
            'windows' : [display = 3]
      END_CASE
      [alarm = on]
      IF [reset == TRUE && password == TRUE]
            THEN [alarm = off ]
END
```

这段伪指令代码显示该程序是一个无限循环。只要没有传感器触发，程序就处于等待状态，什么都不做。一旦有传感器被触发，系统查看对应的闯入地点，然后用显示在七段数码管上的数字0~3来指出闯入点，然后警报器被拉响。该系统还允许进入复位模式，如果用户键入正确的用户密码，系统就可以复位，同时警报器被关闭。

13.3 电梯系统

电梯系统是另一个学生们可以承担的有趣题目。出于降低难度的目的，只考虑设计一个3层的电梯系统。

13.3.1 项目描述

该项目的描述如下。

设计一个基于8051的电梯系统，支持3层升降，包括底层、一楼、二楼。

355

系统由以下两部分组成。

(1) 电梯内部。

图13-1展示了电梯内部的控制和现实面板。其中有3个楼层选择按钮：G，1，2；乘客可以通过它们控制电梯带自己到达目的楼层。开（关）门按钮用来打开（关闭）电梯门。同时，一系列发光二极管（8个）从左到右顺次点亮（每次一个），来指示电梯门的开（关）状态。最左边的LED亮了意味着门已经关闭。最右边的LED亮了则表明门已经完全打开。

(2) 电梯外部。

图13-2展示了电梯的外部的控制和现实面板布局。每层楼都有一个楼层指示器来指示电梯当前是否在该楼层。电梯外等待的乘客通过呼叫按钮请求电梯到达自己

所在的楼层，同时表明目的运行方向（是上楼还是下楼）。

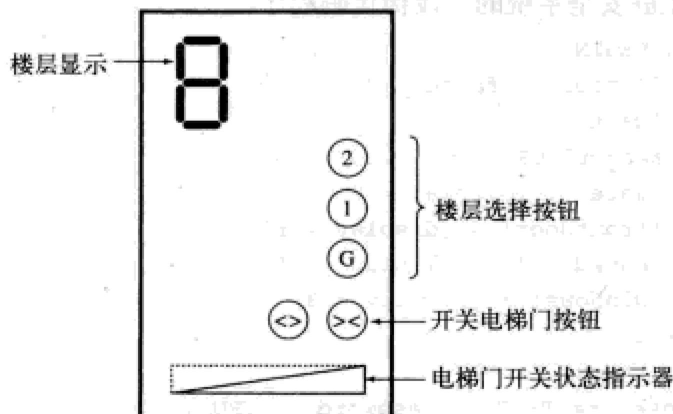


图13-1 电梯内部的控制和显示面板布局

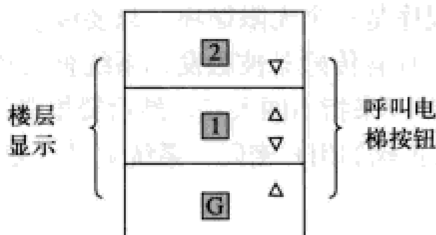


图13-2 电梯外部

13.3.2 系统规格

通读题目描述后，会发现该电梯系统需要如下器件：

- 1个8051微控制器
- 9个开关按钮（内部5个，外部4个）
- 11个LED（门的开关状态指示8个，楼层指示3个）
- 1个七段数码管

356

实际上很清楚，这个项目仅仅包含一些开关和LED，外加一个七段数码管的相互作用。因此，它和前一节中的家庭安全系统并没有太大的不同。唯一的不同是，本项目需要考虑一个完全不同的场景，事件（开门、电梯升降等）发生的次序也比前者更复杂。

13.3.3 系统设计

在这个例子中，系统设计阶段需要决定如何将开关按钮、LED以及七段数码管和8051的端口相连接。因为电梯系统不需要使用外部存储器，所以8051的4个端口都可以用作通用I/O，有很多资源可以利用。

13.3.4 软件设计

电梯系统的最终程序将会是比较复杂的，因为有许多不同的特殊情况需要考虑到，如下所示。

- ❑ 当乘客按下升/降按钮的时候，电梯会正好在乘客所处的楼层吗？
- ❑ 电梯的运行方向和乘客请求运行（即上升或下降）的方向一致吗？
- ❑ 如果不是，那当接近发出请求的楼层时，电梯是会停下来首先满足这个服务，还是忽略它，完成当前的任务再说呢？
- ❑ 电梯服务的原则是优先满足最近的请求，还是尽量维持恒定的方向从最低的请求楼层到最高请求楼层，直到改变方向前，重复这一过程？
- ❑ 如果当前没有请求发出，电梯处于什么状态，在做什么？

解决这些问题需要做出许多选择，之前的学生们提出了各种各样的方案来完成这个课题。一个可能的程序需要参考下面的一段伪码程序来编写：

```
[start at ground floor]
WHILE [1] DO BEGIN
      WHILE [elevator summon == FALSE] DO
            [wait]
      IF [summoning floor == current floor]
            THEN BEGIN
                 [door = open]
                 WHILE [open door button == TRUE] DO
                       [wait]
                 [door = close]
            END
      [direction = up]

  BEGIN
        WHILE [ [summoning floor != current floor] &&
              [requested floor != current floor]] DO BEGIN
                   IF [current floor == TOP_FLOOR]
                         THEN [direction = down]
                   IF [current floor == GND_FLOOR]
                         THEN [direction = up]
                   IF [direction == up]
                         THEN [current floor = current floor + 1]
                         ELSE [current floor = current floor - 1]
                   [floor indicator = current floor]
                   [floor display = current floor]
        END
        BEGIN
             [door = open]
             WHILE [open door button == TRUE] DO
```

```
            [wait]
        [door = close]
        END
    END
END
```

这里，“当前楼层(current floor)”就是电梯目前所处的楼层，“呼叫楼层(summoning floor)”就是电梯外有等待乘客按下呼叫按钮的楼层，同时，“请求楼层(requested floor)”指的是电梯内的乘客通过楼层请求按钮希望前往的楼层。

程序是一个无限循环，电梯最开始在底层，等待有需要的乘客（在他们的楼层）召唤。当有人按下呼叫按钮时，程序会检查电梯是否已经在该楼层。如果正好在，就打开电梯门，等待一段时间，直到开门按钮不再被按下，电梯门被关闭时，才会开始上升。

当电梯从一层向另一层运行时，程序会不断检查是否有人在当前楼层按下召唤按钮，电梯内是否有乘客前往当前楼层。如果没有，电梯会继续移动至下一个楼层。如果当前楼层已经是最顶层或者最底层，接下来电梯将会沿向相反的方向运行。与此同时，电梯外部的楼层指示器和楼层内部的楼层显示器都会适时地调整到相应的楼层显示。

如果当前楼层正是呼叫楼层或者请求前往的楼层，那样，电梯就会打开电梯门，等待乘客释放开门按钮之后，关闭电梯门。

13.4 井字游戏

对学生而言，开发游戏是一个更富有挑战性的课题，程序本身具有的人工智能(AI)允许人机对战或者双人对战。基于8051的井字游戏就是这样的一个题目。

358

13.4.1 项目描述

首先，简述项目描述和游戏规则。

设计基于8051的井字游戏，主要包括以下两部分。

(1) 井字游戏环境。

游戏开始前，选手要从3种游戏模式（人对人，人对微控制器，微控制器对人）中选择一种。游戏模式通过P1.3和P1.4引脚设置，具体的细节可参考表13-1。

表13-1 游戏模式

模　式	P1.3(M1)	P1.4(M0)
0—人对人	0	0
1—人对微控制器	0	1
2—微控制器对人	1	0

一个3×3的井字面板（如图13-3所示）用来指示哪个格子已经被选手选中。选中的格子用黄色（选手1）点亮或者用红色（选手2）点亮。当无论横、竖或者对角线方向有三个格子点亮为同一种颜色，即被同一选手选中，就赢得了比赛。反之，如果所有的格子都被选中，也没有连成一组（3个）的同样颜色的格子，那么双方以平局收场。9个格子中的每一个都由黄和红两种颜色的LED组成，所有的LED都通过P0、P1和P2端口与8051连接，具体的对应关系见图13-4。

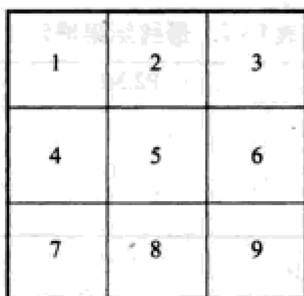


图13-3　3×3井字面板

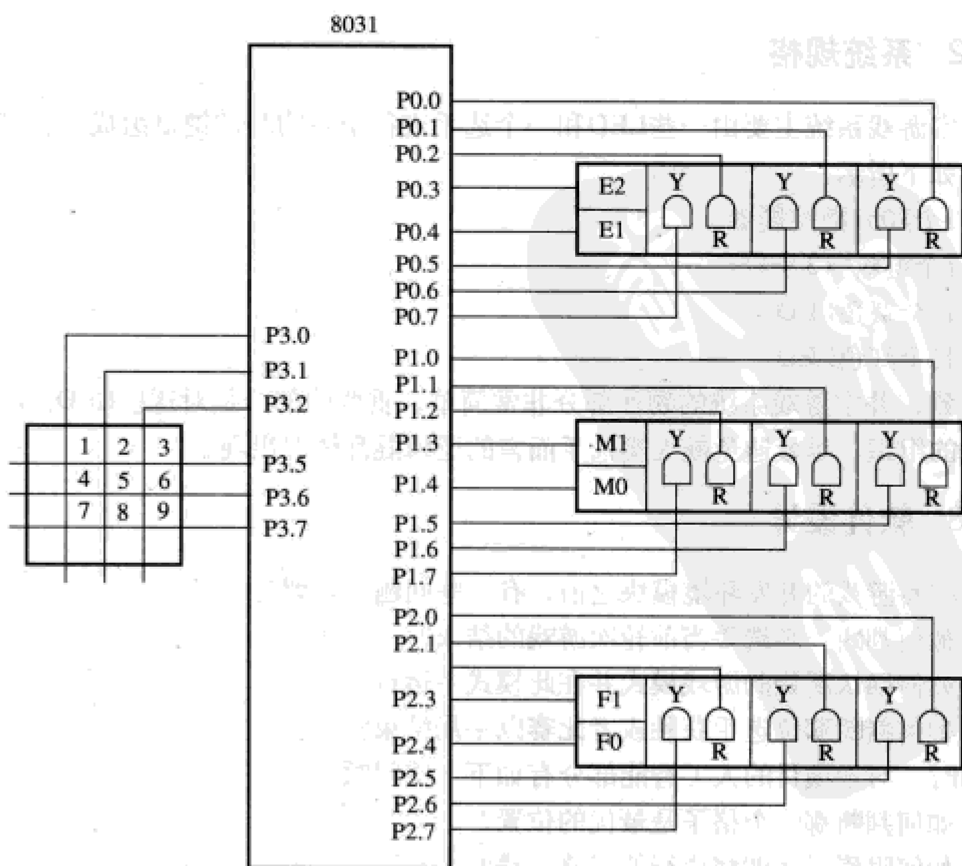


图13-4　游戏面板

每位选手轮流通过一个3×3矩阵键盘来选择某个特定的格子。8051通过P3端口与键盘进行连接。1号选手通过设定P0.4（E1）点亮轮次结束指示灯来结束自己的轮次，同样的，2号选手通过设定P0.3（E2）点亮相应的轮次结束指示灯结束他的轮次。

当游戏结束时，比赛的结果会通过连接P 2.3和P 2.4的LED显示，具体的细节可见表13-2。

表13-2　最终结果显示

模　式	P2.3(F1)	P2.4(F0)
选手1（黄色）胜利	1	0
选手2（红色）胜利	0	1
平局	1	1

(2) 井字游戏的人工智能模块。

人工智能模块应该能够做到扮演1号或者2号选手进行比赛，并且应该尽最大努力来战胜人类选手。

359

13.4.2　系统规格

井字游戏系统主要由一些LED和一个选手进行游戏的3×3键盘组成。详细的器件清单如下所示：

- 1个8051微控制器
- 1个键盘（3×3）
- 11个黄色LED
- 11个红色LED

可以看到，井字游戏系统的硬件部分非常简单，重要的部分是对这些LED清晰的合乎逻辑的组织，尽量避免就人类选手而言的逻辑混乱情况出现。

360

13.4.3　软件设计

在编写游戏的开发环境模块之前，有一些问题必须留意。

- 如何判断一名选手当前轮次游戏的结束？
- 如何确认所处的游戏模式并在此模式下运行？
- 如何判断哪位选手获胜或者比赛以平局结束？

同时，对于项目的人工智能部分有如下一些问题。

- 如何判断哪一个格子是最优的位置？
- 如何阻碍对手即将成行的三格一线？
- 有没有一些高级和精妙的战术来骗取对手选择对你有利的格子？

❑ 人工智能的游戏策略是应该固定还是随当前形势的变化而改变？

井字游戏环境的一段可能的伪代码程序在下面给出：

```
WHILE [input == FALSE]
       [wait]
[current player = yellow]
CASE [mode] OF
       '0': BEGIN
              WHILE [1] DO BEGIN
              [check current player color]
              WHILE DO [player input == FALSE]
                     [wait]
              [selected box == player color]
              IF [3-in-a-row == TRUE]
                     THEN BEGIN
                            [display winner]
                            [end game]
                     END
              IF [all boxes selected == TRUE]
                     THEN BEGIN
                            [display draw]
                            [end game]
                     END
              END
       END
       '1': BEGIN
              [current player = human]
              WHILE [1] DO BEGIN
                     [check current player color]
              IF [current player == AI]
                     THEN [call AI]
              WHILE DO [input == FALSE]
                     [wait]
              [selected box == player color]
              IF [3-in-a-row == TRUE]
                     THEN BEGIN
                            [display winner]
                            [end game]
                     END
              IF [all boxes selected == TRUE]
                    THEN BEGIN
                            [display draw]
                            [end game]
                     END
                     current player = other player]
              END
 END
 '2': BEGIN
        [current player = AI]
        WHILE [1] DO BEGIN
              [check current player color]
              IF [current player == AI]
                     THEN [call AI]
```

361

```
                WHILE DO [input == FALSE]
                      [wait]
                [selected box == player color]
                IF [3-in-a-row == TRUE]
                      THEN BEGIN
                            [display winner]
                            [end game]
                       END
                IF [all boxes selected == TRUE]
                      THEN BEGIN
                             [display draw]
                             [end game]
                      END
                [current player = other player]
                END
          END
    END_CASE
```

程序等待选手给出开始井字游戏的输入命令。一旦检测到开始输入，程序就对当前选手选择的游戏模式进行判断。如果是模式0（人对人），那么程序就会进入一个无限循环，同时查看谁是当前选手，等待当前选手完成他的轮次。一个输入信号的发出就意味着某个特定格子被选中，程序就会将该格子的颜色改变成和当前的选手的颜色一致。当前选手完成他（她）的轮次时，程序就会检查是否有3个相同颜色格子排成了一线（无论是横、竖方向还是对角线方向）。如果有这种情况出现，系统显示胜利者标志并结束比赛。否则当程序检查到所有格子都被选中，意味着比赛是平局，比赛结果也会立即显示。

362

如果是模式1（人对微控制器）被选择，人类选手首先开始比赛。程序进入一个无限循环，检测当前选手。如果当前选手是微控制器，就会调用人工智能函数来计算选择最优的格子。无论微控制器或者人类选手，当一个格子被选中时，将显示那位选手的颜色。程序进一步检查是否有3个颜色相同的格子成一线，出现上面的情况意味着检测到了获胜者，显示结果，并结束游戏。

模式2（微控制器对人）和模式1类似，唯一的不同就是比赛由微控制器先开始。

下面给出了一段井字游戏的人工智能程序：

```
IF [opponent's 2-in-one row == TRUE]
      THEN [block opponent's chance]
ELSE BEGIN
      [calculate best box position]
      WHILE [box == taken] DO
            [calculate next best box position]
      [select box]
      END
```

这个人工智能程序看起来相当简单。主要原则就是阻碍对手形成三格一线的企图，因此如果检测到两个格子连成一线的情况，人工智能程序会选择相应的第3个格子

来达到阻止对手获胜的目的。接着，如果没有发现有两个格子排成一线的情况，那意味着还很安全，有很大的灵活性来按照自己的计算选择最优的格子。如果最优的格子已经被选中，程序会计算第二优的位置，如此一直到没有被选中的格子，立即选中它。这看起来似乎很简单，但是编写人工智能程序最重要的任务是计算最优格子的位置。这才是从很多人工智能程序中辨别优劣的重点。

把整个软件分成两个部分将会使得整个编程任务更容易一些。这也是个有趣的**模块化编程**的练习，一个程序被分成几个模块，每个模块由不同的程序员编写。一旦编程完成，再把这些模块组合成一个完整的程序。模块化编程允许一个程序的不同部分同时开发，非常适合大型复杂程序项目的编写。

13.5 计算器

现在，计算器的应用非常普遍，以至于人们常常忽略它的存在。但是，作为基于8051系统设计和接口的进一步练习，基于8051的计算器是个相当不错的题目。当然，为了简化问题，只考虑最基础的运算操作。

13.5.1 项目描述

项目描述如下所述。

设计一个基于8位8051微控制器的计算器系统，可以进行一些简单的运算：

- 加法
- 减法
- 乘法
- 除法
- 逻辑与
- 逻辑或
- 逻辑取反
- 逻辑异或
- 平方
- 平方根
- 立方
- 立方根
- 倒数
- 求幂
- 阶乘
- 模运算

- ❑ 百分数
- ❑ 圆周率π
- ❑ 基本的存储运算

计算器应该能够支持16位应答，答案显示在LCD(液晶显示器)或者七段数码管上。另外还需要设计一个矩阵键盘供用户输入信息。

13.5.2 系统规格

计算器需要一个矩阵键盘进行用户输入，还有一个LCD或者七段数码管来显示计算结果。除此之外，至少还需要一个按钮来做计算器的开关，一个相应的LED来显示计算器的开关状态。所以，该系统需要的器件如下：

- ❑ 1个8051微控制器
- ❑ 1个矩阵键盘
- ❑ 1个LCD/5×7点阵液晶显示模块
- ❑ 1个LED
- ❑ 1个按钮开关

13.5.3 软件设计

控制这个系统的软件与Windows或者MS-DOS等操作系统上的计算程序很类似。在有操作系统的机器上，所有算术和逻辑运算以及输入输出设备间的交互作用都由操作系统来控制。在写这段程序之前，下面的一些问题需要考虑。

364

- ❑ 如何判断用户按下的按键是1个数字还是1种算符（如+或－等）？
- ❑ 如何处理运算优先级？
- ❑ 是否允许累加运算？
- ❑ 计算器进行运算的数字是有符号数还是无符号数？
- ❑ 计算器是否支持小数运算？
- ❑ 如何实现每种类型的算术运算或逻辑运算？
- ❑ 是否采用查表方法？

一个可行的8051计算器基本可以参考如下的伪代码程序：

```
WHILE (1) DO BEGIN
      WHILE [keypress == FALSE]
             [wait]
      CASE [keypress] oF
             'numeric': BEGIN
                 [save numeric value]
                 [format for 7-segment display]
                 [send to 7-segment display]
```

```
                    END
            'operation': BEGIN
                    [determine operation]
                    [save operation]
                    END
            'memory operation': BEGIN
                    [determine operation]
                    [perform operation]
                    END
            'equal': BEGIN
                    [recall saved numeric values and operations]
                    [perform operations]
                    [format result for 7-segment display]
                    [send to 7-segment display]
                    END
        END_CASE
    END
```

程序等待一个不确定的按键被按下，当监测到有“按键”事件发生时，它会判断该键对应的是一个数字、算术或逻辑运算，还是一个存储操作或者一个“=”。当前两者被检测到时，程序会明确核对是哪个数字或者哪个运算符，并保存它们。数字会被格式化并送入七段数码管显示。如果是存储操作请求，程序会立即判断是什么存储操作并提供相应的服务。最终，如果有“=”被按下，这意味着用户想看到计算结果，之前所有存储的数值和运算符都被调用，并根据优先级进行运算。结果格式化后送入七段数码管显示。

13.6 微型老鼠

当把8051微控制器（作为大脑）和一些传感器（作为眼睛、耳朵等）还有一些电动机（手和脚）连接起来时，这就是一个初级的机器人了。**微型老鼠**正是这样的一个机器人，它模仿了一只试图走出迷宫的老鼠。走迷宫需要微型老鼠从迷宫的一头开始通过不断地尝试、犯错、记忆，慢慢地找出正确的道路到达迷宫的另一端。一旦找到正确的路径到达迷宫的另一端，当它下次被放入迷宫时，就能沿着这条路走出来。微型老鼠的比赛经常会在世界各地方举办，非常流行。本节将探讨一个微型老鼠项目。

13.6.1 项目描述

按照惯例，首先描述一下这个项目的细节。

设计一个基于8051微控制器的微型老鼠，使其具有前进、左转、右转的能力，也可以做U字形转身。它有能力足够快地走出迷宫，一旦走出迷宫就能将正确的路

径存储在其存储器中。

13.6.2 系统规格

为了能够前进和转弯，应该分别在微型老鼠的左右轮子上各安装一个电动机。它应该能知道自己是否进入了一个死胡同，或者在自己想要转弯的方向上，左侧或右侧有一面墙堵在那里。因此，微型老鼠需要一些传感器，有两个红外传感器就足够了。大概还需要一些LED来标示微型鼠当前想做什么，是想前进、左转、右转，还是做U字转身。此外，微型鼠还需要一些较大的外部存储器来记忆之前走过路线。因此，需要如下器件：

- ❑ 1个8051微控制器
- ❑ 2个电动机
- ❑ 2个红外传感器
- ❑ 1个64KB RAM
- ❑ 2个LED

13.6.3 系统设计

系统设计阶段最主要的部分是决定在哪里安装红外传感器。典型的红外传感器由并排放置的两部分组成：一个红外LED负责发射红外线和一个红外探测器（光电二极管）负责监测红外线的入射。如果红外传感器的对着的方向有墙，红外LED发射的红外线就会被反射回来，然后被红外探测器感应到。否则，如果这一侧没有墙，就不会产生反射光，也不会有红外线被探测到。这就是微型鼠如何判断在某一个方向上是否有墙的基本原理。

366

其中一个传感器应该放置在小老鼠的前面，用来监测前方是否有墙阻碍其前进。另一个传感器应该被安置在小老鼠的左侧或者右侧。假设已经决定把它放在左侧，如图13-5所示。

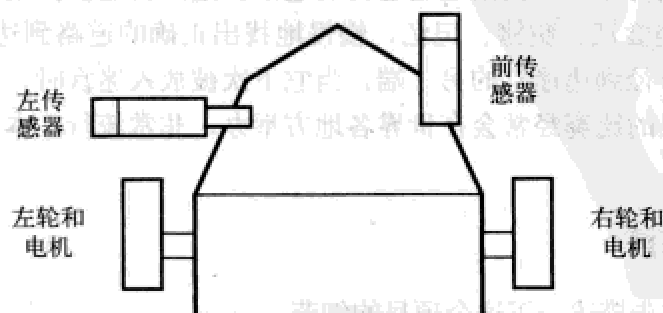


图13-5 微型老鼠的俯视图

另一个问题是：如果仅在微型老鼠的一侧放置传感器，怎么能够探测到另一侧

是否有墙存在？回答是：并不是非得通过直接探测得到结果。下面通过例题13-1~13-3来考虑这个问题。

例13-1 微型鼠遇到左拐角

微型鼠如何知道自己已经到了一个左拐角了呢?

答案：

前传感器探测到墙，而左传感器没有探测到。

讨论：

在这个例子中，微型老鼠的前传感器发现有墙阻挡其前进方向。同时，左侧传感器并没有发现有墙存在。因此它得知已经遇到了一个拐角，不能再继续前进，只能左转。

当然，还可能有一种情况，即微型鼠遇到的是一个T字形路口，它既可以左转也可以右转。但无论如何，左侧肯定是没问题的，那么微型鼠就应该首先尝试左侧的路径，稍后再尝试右边的路径。

例13-2 微型鼠遇到一个右拐角

微型鼠如何得知自己已经到了一个右拐角?

答案：

前探测器和左传感器都探测到了墙。

讨论：

既然两个传感器都探测到了墙的存在，微型鼠既不能前进也不能左转。因此，它猜出自己的右边应该没有墙，转身向右。

可是，微型鼠遇到了一个死胡同也是很可能的，它的右侧也是一道墙。下个例子中将探讨这个问题。

例13-3 微型鼠遇到一个死胡同

微型鼠如何得知自己已经遇到了一个死胡同?

答案：

首先，前侧和左侧都探测到了墙。然后，转向右侧，前探测器又探测到了墙。

讨论：

首先，前探测器和左探测器都探测到了墙，所以微型鼠右转。可是，右转之后，他的前探测器仍旧探测到了墙，这只能是遇到了1个死胡同。这种情况下，它应该第二次右转，这样它实际上已经向右转了两次。这本质上就是1个U字弯。它现在就可以继续前行，沿着刚才来到死胡同的路线反向离开。

13.6.4 软件设计

微型鼠的软件设计是比较复杂的，需要一些深入的思考，包括下面的几个要点：

- 微型鼠如何探测分辨出右拐角、左拐角和死胡同？
- 微型鼠如何进行左转或者右转？
- 应该按照什么顺序检测所有可能的路径？首先检验最近的支路还是最远的支路？
- 微型鼠如何存储之前走过的路径？

微型鼠的参考伪代码程序如下：

```
WHILE [1] DO BEGIN
        [left motor = right motor = forward]
        IF [front sensor == wall]
                IF [left sensor == no wall]
                      THEN BEGIN
                             [left  motor =  backward]
                             [right  motor =  forward]
                      END
                      ELSE BEGIN
                             [left  motor = forward]
                             [right motor = backward]
                             IF [front sensor == wall]
                                  THEN BEGIN
                                         [left motor = forward]
                                         [right motor = backward]
                                  END
                      END
        [memorize path]
END
```

关于探测拐角以及死胡同，在例13-1~13-3中都已经讨论过了。关于左转和右转的问题，其实很容易解决。如果微型鼠要前进，两个电动机都向前转动即可。如果微型鼠想左转，那么左侧电动机向后转动，而右侧电动机向前转动。同样，如果欲右转，令左侧电动机向前转动，右侧电动机向后转动。 368

上面这段伪码程序是一个无限循环，且初始状态是前进。如果微型鼠探测到前面有墙，它会继而探测在自己的左侧是否有墙。如果没有，就左转；有的话，就右转。右转之后，它会探测前方是否有墙，如果仍旧有，就是一个死胡同，它会再一次右转。现在微型鼠已经可以处理所有的情况了，它可以继续前行。在所有这些过程中，微型鼠都会存储它曾经走过的路线。

微型鼠编程中最大的挑战就在于：存储路径。如何通过存储路径来保证当下次

遇到同样的T字路口或者十字路口时，它不会再次选择一条死路？这将会是一个需要深入思考和丰富的设计经验才能完成的课题。

13.7 足球机器人

前一节讨论了8051微控制器如何作为一个类鼠形机器人的大脑，使其可以在迷宫中寻找道路并最终走出迷宫。实际上，8051可以控制任何机器人，是被机器人爱好者作为机器人大脑的最流行的微控制器之一。机器人竞赛在全球范围内都有举行，盛况空前。参与其中的机器人可能被要求完成各种各样的任务，从做简单运动到完成非常复杂的任务，如爬楼梯等。本节将探讨如何用8051控制足球机器人。

13.7.1 项目描述

设计一个基于8051的机器人可参加的、持续10分钟的机器人足球比赛。在每场比赛中，代表蓝队和红队的两个机器人被放置在一个铺灰色的地毯的场地中，同时，场地中随机放置着直径15mm的红色和蓝色足球各一个。一个共用的球门被放置在场地的一侧，并用灯光照亮。每个机器人都要去探测足球的位置，并判断球的颜色，去搜集和自己同颜色的球，但是一次只能放一个球。一旦机器人收集到一个球，它就会寻找到共用的球门并把球踢进去。

13.7.2 系统规格

和微型鼠类似，机器人需要一些由电机驱动的轮子，也需要一些光传感器（LED和光电二极管对）来探测足球的位置和颜色，外加一些接触传感器（如开关）来探测墙和对手。机器人是独立存在的（即无上位机控制），因此需要安装非易失性存储器。在这个例子中，因为EPROM允许根据意愿重写，所以EPROM是最好的选择。需要的器件列表如下：

- ❑ 1个8051微控制器
- ❑ 2个电机
- ❑ 4个轮子
- ❑ 2个光传感器（LED和光电二极管对）
- ❑ 4个开关
- ❑ 1片EPROM

13.7.3 系统设计

在设计这个机器人系统时，在哪里放置光传感器和接触传感器是非常重要的。例如，判断颜色的光探测器需要很好地屏蔽掉外部环境的干扰，否则就可能对颜色

的探测带来不利的影响。尤其是当机器人非常接近球门的时候，明亮的球门灯会干扰传感器上的LED发出的光并导致混淆。

另外一个设计重点是用来“踢球”的撞击装置。有许多不同的踢球方法，观察不同的学生想出的不一样的主意，也是一种乐趣。有些人可能会想用电动机来带动一个摆杆来击球，就像棒球运动员那样用球棒击球。这种技术的局限性在于与弹力相比力道小了很多。因此，一些人更愿意在球杆上绑上一段橡皮筋，然后用一个电动机来把橡皮筋拉长。当机器人准备踢球时，电动机移动到一个可以使橡皮筋被释放的位置，将橡皮筋释放，使球杆摆向足球，将球“踢出”。

13.7.4 软件设计

关于机器人软件的一些问题如下。

- 机器人如何探测墙和机器人对手？
- 这个机器人可以转动吗？
- 当机器人到达一个角落时，它是右转还是左转？一直向一个方向转向还是交替向不同的方向转？或者随机确定方向转动？
- 这种随机性如何实现？
- 机器人如何判断球的颜色？
- 机器人如何侦测球门的方向？
- 机器人如何踢球？
- 机器人采用什么策略来踢球？更富有攻击性，还是偏向防守？如何在比赛中切换策略？
- 机器人在场内如何移动？是随机走还是按照某种路线走？如果是按照固定路线走的话，那么如何确定最好的路线来保证可以覆盖尽可能大的面积？

下面给出的是一段控制蓝队机器人的伪码程序：

```
WHILE [1] DO BEGIN
        [walk in spiral pattern]
        IF [contact sensor == TRUE]
            THEN BEGIN
                    [reverse]
                    [turn left or right]
            END
IF [light sensor == BLUE]
        IF [balls collected < 1]
            THEN BEGIN
                    [collect ball]
                    [sense goal lights]
            IF [goal detected == TRUE]
                    THEN BEGIN
                            [stop]
```

```
                        [kick]
                  END
            END
    IF [light sensor == RED]
        IF [time left == 5 minutes]
            IF [balls collected < 1]
                THEN BEGIN
                        [collect ball]
                        [go straight forward]
                        IF [wall detected]
                            [release ball]
                END
    END
```

这段程序指挥机器人按照螺旋路径来行走，大部分人相信这是一种可以覆盖尽量大场地的好方法。当接触传感器返回真值时，意味着机器人正向一面墙或者对手走去，此时机器人转身，然后左转或者右转，远离障碍物，继续它前进的脚步。如果光传感器探测到一个自己对应颜色的蓝色球，它会检查是否已经有一个球在身上。如果没有，机器人就会把球收集上车，然后立即寻找球门。一旦探测到球门，它停下来，把球踢进去。如果光传感器探测到了一个红色球，机器人会检查比赛是否只剩5分钟，如果是，就切换为一个更具防守性的策略：它收上对手的球，大步向前冲，一直到撞倒了墙的时候，就把球放下。因为紧紧靠着墙的球通常是很难被收上车的。这就是蓝队机器人针对对手的防守策略。

13.8 智能卡应用

微控制器越来越多地应用在**智能卡**上，智能卡将存储器和微控制器集成在了非常有限的物理空间内。想象一下一台计算机小到用一片可以装进钱包的塑料片来做包装，这就是智能卡。智能卡被用于身份证、驾照、护照信息以及电子货币。这些智能卡包含个人信息甚至一些敏感的金融信息，并且都是要在公共场合中使用的。因此，保证这些信息不被未授权者读取和篡改是非常重要的。下文简要描述了基本的安全概念，让读者更好地理解如何保护智能卡中的信息。

371

13.8.1 基本安全概念

信息安全通常需要保证信息的**机密性**、**可认证**和**完整性**。机密性是要确保只有经过授权的当事人可以读取和浏览信息；认证验证当事人的身份；完整性允许人们来判断信息是否在未经许可的情况下被篡改。

所有这些都可以通过**加密**、**数字签名**、**消息认证**这些技术来实现。加密是通过把一些机密信息转换成非常难懂的形式来保护信息不被读取的过程。标准加密方法是数据加密标准（DES）和它的衍生物——Triple DES。顺便提一下，后者被广泛

运用在许多当前自动取款机（ATM）上，以保护用户的个人身份码（PIN）。

同时，数字签名在概念上和它对应的手写签名非常相似。但它取代了手写签名或者指纹，采用只有几比特的保密数据来验证当事人的身份。消息认证通过几比特的保密数据来实现，这些保密数据是通过对消息执行一种特殊操作而得到的**消息认证代码**（MAC），这非常类似于校验和的方法，故消息认证被用于检查消息是否已经被篡改。MAC只有在知道秘密数据的前提下才能获得，这就保证了未经授权方不可能修改信息而不被察觉，因为每个MAC都不同。

13.8.2 项目描述

回顾上一章的难点部分，这里讨论一个非常简单的加密方法——凯撒密码。现在考虑用一个有一些复杂但具有通用性的方法，即**多字母替换法**。尽管如此，这种加密方法在现代已经不再用来保护信息，因为它非常容易被计算机破解。为了便于说明，本节将仍然讨论这种方法，只是在本章末尾的难点部分会接触一些现代的安全方法。

多字母替换法与凯撒加密法非常相似，区别是后者是将原始明文字母表固定左移三个位置，而多字母替换法向左移动的位置数是一个变量，而不是固定的。例如，左移10个位置，明文字母表会被相应下面的密文字母表所替代。

明文：A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

密文：K L M N O P Q R S T U V W X Y Z A B C D E F G H I J

所以消息THIS IS SECRET被加密为DRSC SC COMBOD。

设计一个基于8051微控制器的智能卡，可以运用多字母替换法进行加密。空字符结束消息存储在EPROM中1234H地址开始的存储空间。

13.8.3 系统规格

智能卡系统是相当简单的，仅由1个8051微控制器和1片EPROM组成。EPROM既可以是外部存储器，也可以是微控制器内部自带的存储器均可。如果是微控制器内部自带存储器，就用8071代替8051。另外可能会选择添加一些外部器件，例如1个LCD来显示明文和密文信息。需要的器件如下：

372

- ❑ 1个8051微控制器
- ❑ 1片EPROM
- ❑ 1个LCD

13.8.4 软件设计

在写这个加密软件的时候，需要注意下面一些问题。

- ❑ 移位的位数如何确定？是随机选择还是允许用户指定？

❑ 是否某些移位位数应当被禁止？例如，移位0、26、52或者其他26的倍数，因为这些位数都会导致明文和密文完全一样，也就意味着没有加密。

相应的伪码程序：

```
BEGIN
    [determine number of shifts]
    [location = 1234H]
    REPEAT
        BEGIN
            [get character from location]
            [encrypt character]
            [store back in location]
        END
    UNTIL [character == NULL]
END
```

软件可以采用顺序结构。首先，确定移位个数，首位地址设在1234H。程序从当前地址取值，用多字母替换法加密，将加密过的字符储存在当前地址，因此覆盖掉了之前存放的字符。该过程不断重复直到空字符的出现，表明消息结束。

小结

本章讨论了一些高级的基于8051微控制器的项目，这些对学生们来说是一个挑战，他们对此也会非常感兴趣的。可行的方案是按照循序渐进的方式进行本章的学习，从对项目描述的仔细研究开始，到系统需求的规格以及器件列表。然后还有一部分软件设计的内容，包括编程需要考虑的问题和相应的描述软件的伪码。

373

习题

13.1 画出13.2节中家庭安全系统的原理方框图，包括8051与警报器、传感器、七段数码管、外部存储芯片之间的所有连接。用C或者汇编语言编写控制该系统的软件。

13.2 画出13.3节中电梯系统的原理方框图，包括8051和LED、开关以及七段数码管之间所有的连接。

13.3 观察几个电梯系统。这些电梯系统按照什么顺序执行请求？为什么采用这样的顺序？

13.4 画出13.4节中井字游戏系统的原理方框图，包括8051与键盘、LED和开关之间所有的连接。用C或者汇编写出此系统的软件。

13.5 画出13.5节中计算器项目的原理方框图，包括8051与键盘、显示器、LED和开关之间所有的连接。

13.6 用汇编语言子例程或者C函数来实现基于8051计算器项目描述中列出的任意两种算术运算。

13.7 画出13.6节中微型老鼠项目的原理方框图，包括8051与电机、传感器、LED和外

部存储器之间的所有连接。

13.8 回答下列关于微型老鼠系统编程中的问题。

a. 测试所有可能的路径。采用哪种次序可以测试完全？最近的支路优先还是最远的支路优先？为什么？

b. 存储之前走过的路径。你认为达成这一目的的最好方法是什么？为什么？

13.9 当利用智能卡制作身份证件时，通常期望个人照片等信息也被嵌入卡内。但是，智能卡的一个局限性就是它的存储器比较小。调查研究当前所采用的智能卡照片嵌入技术。

13.10 一天在教室里，你碰巧发现一张折着的纸条，上面是某位同学写给另一位异性同学的话，"一定很有意思"，你这样想着，同时笑容也绽放在了你的脸上，可当你看到上面写着：

atih spit idcxvwi

你的笑容一下子变成了疑问。前几天你在班上刚刚学习了凯撒加密法。你猜测你的同学一定用的这种方法。当你坐下试图解密消息时，你发现这并不是凯撒加密法加密过的。不过，你坚信一定是某种类似的方法，不是像凯撒加密法那样右移三个位置。你必须知道你同学采用的加密法到底平移几个位置。一定是采用的多字母替换法。一个创意突然在你脑中展开，用8051来解密它。请写一段基于8051的程序解密这段消息。

374

提示：你需要采用被称为**字典攻击**的手段：尝试所有可能的移位数，逐字逐句地与词典比较每种解密后的译文，看是否获得匹配。一旦三个词都匹配，你很可能已经解密成功。假设有一个有1000条词目并且包含原始消息的单词，原始消息中的单词被数组中不断尝试被其他字符所替代：

```
Char * dictionary = {"a","able","about",…};
```

当dictionary[0]给出一个字符a时，dictionary[1]会给出able，以此类推。

假设所有字母都是小写，相应的ASCII码如表13-3所示。

表13-3 部分ASCII码对应表

字符	ASCII码表		
	十进制	十六进制	二进制
a	97	61H	01100001
b	98	62H	01100010
c	99	63H	01100011
d	100	64H	01100100
e	101	65H	01100101
f	102	66H	01100110
g	103	67H	01100111
…	…	…	…

要求写出这个程序的伪指令代码。

13.11 写出习题13.10相应的C语言程序。然后，用它来对该加密消息进行解码。

13.12 高级密码标准（AES）是最近出现的一种加密标准，很可能在将来的安全应用中替代DES和triple-DES。研究AES的工作原理，写一段8051程序(汇编或者C均可)来实现这种加密方法。关于AES的简短描述在附录J中给出。

375

第14章 8051的派生产品

14.1 本章简介

自从MCS-51™系列微控制器IC产品问世以来，更新更先进的版本不断出现。这些由8051派生出来的微控制器都带有附加存储器，增加了ADC和DAC这样的输入/输出端口，还有其他扩展资源。本章将简要介绍一些这类微控制器。

14.2 MCS-151™和MCS-251™

英特尔公司推出了更高版本的MCS-51™系列产品。这些MCS-51™在性能上提高了5倍，随后又开发出了性能提高了15倍的MCS-251™系列。MCS-251™系列有一个能有效提高8051 C语言程序设计效率的先进结构，其扩展指令集包括16位和32位的算术和逻辑指令，增加了一个提高存储器容量的功能（见表14-1）。**一次性可编程（OTP）只读存储器**指向类似EPROM的程序存储器，但是该存储器没有可用于擦除内容的石英玻璃窗口。这就降低了封装成本，同时也可防止OTP只读存储器中的数据被紫外光擦除，因此，它只能用于一次性编程。无ROM的版本也可以使用OTP ROM。

表14-1 MCS-251™IC的比较

器件型号	片上程序存储器	片上数据存储器
80251SA	8KB ROM/OTPROM	1KB
80251SB	16KB ROM/OTPROM	1KB
80251SP	8KB ROM/OTPROM	512B
80251SQ	16KB ROM/OTPROM	512B
80251TA	8KB ROM	1KB
80251TB	16KB ROM	1KB
80251TP	8KB ROM	512B
80251TQ	16KB ROM	512B

14.3 带有闪存和NVRAM的微控制器

第2章介绍了拥有不同类型片上程序存储器的MCS-51™产品，包括无ROM的8031，带有片上只读存储器的8051和带有片上EPROM的8751。

带有EPROM程序存储器的8751微控制器允许采用编程器更新其中的程序，即便如此，也必须先用紫外光EPROM擦除器擦除其中的旧程序，之后才能再次烧录新程序。与此相比，由8051派生出来的8951微控制器带有一个片上闪存，它的基本功能类似于EPROM，但是其中的程序可以由EPROM编程器进行电擦除。因此，8951微控制器无需额外的紫外光EPROM擦除器即可进行旧程序的擦除。Atmel公司①就是一个生产此类8951的制造商。

除了具有闪存的派生微控制器，还有一类如Maxim Integrated Product's DS5000②的微控制器，它们为程序存储器配了一个NVRAM。NVRAM优于闪存的特点是，允许在程序存储器中一次一个字节地修改程序，而不必在重新编程前全部擦除原有程序。DS5000的NVRAM允许程序通过PC机的串行端口重新载入程序，而不需要单独的PROM编程器。

14.4 带有ADC和DAC的微控制器

8051派生系列产品的最基本特性之一就是构建了片上模数转换器（ADC）和数模转换器（DAC），它可在8051与输入/输出端口为模拟信号的设备连接时，省掉了外部ADC和DAC芯片。例如，由Siemens公司制造的带有10位ADC的SAB80C515A型微控制器就是此类型的微控制器。由Atmel公司汇集了8051系列的优点生产出更高级的派生产品，而且带有MP3播放器功能，其中的AT89C51SND1C和AT89C51SND2C两个型号甚至带有当前流行的能与PC机连接的USB1.1接口。

14.5 高速微控制器

8051运行速度为每个机器周期12个时钟周期。而像MCS-151™和MCS-251™这类的高速派生产品能以每个机器周期2个时钟周期的速度运行，因此在相同时间内能执行更多的指令。

378

Maxim公司也生产各种类型的8051派生产品，包括高速微控制器、网络微控制器和保密类微控制器。与8051的12个时钟周期相比，其高速微控制器以每个机器周期4个时钟周期的速度运行，而其超高速微控制器则能以每个机械周期1个时钟周期的速度运行，如DS89C420型号的微控制器。其他优于8051的地方还包括拥有更多的中断源和增加了存储器的容量。

14.6 网络微控制器

由Maxim公司制造的网络微控制器能支持各种网络协议，如以太网和现场总线

① Atmel 公司，地址：2125 O'Net Drive, San Jose, CA 95131。

② 美信（Maxim）IC 产品公司，包括Dallas半导体公司。

(CAN)。其他网络微控制器例如Philips[1]公司生产的83751能支持I²C网络接口，而由Standard Microsystems[2]公司生产的COM20051可支持ARCNET令牌环网协议。这些网络协议允许将几个微控制器和其他处理器连接在一起构成网络，进行分享和交换数据。Atmel公司生产的ATWebSEG-32型微控制器也是一个8051的派生产品，它可支持国际互联网（TCP/IP）和以太网协议。

14.7 保密类微控制器

在前面的章节里介绍了8051如何作为智能卡中的"大脑"使用。还讨论了如何通过软件加密术进行机密信息的保护。实际上，专用的保密硬件同样能完成加密的任务。一些8051的派生产品（如Maxim公司生产的保密类8051微控制器）就是为实现这个目的而开发出来的。

能执行加密、数字签名和消息验证的智能卡中的保密系统被称为**公匙基础设施**(PKI)。PKI一般嵌入到保密类微控制器中，并且由与这些微控制器相连的外围硬件设备来支持。例如，Maxim公司生产的DS5240带有一个可支持模数运算操作的算术运算加速器（MAA），在PKI领域得到了广泛的应用。其他一些保密微控制器（例如Maxim的DS5000）支持通过硬件加密程序后再载入到程序存储器中。采用这种加密方法的程序即使在智能卡被未授权的程序摧毁并读取到这些程序的情况下，攻击者也无法理解程序的意思。

小结

本章介绍了一些8051的派生产品。这些增强版的8051具有性能更优异的片上存储器、大存储容量、高速、支持含模拟I/O端口的设备、网络以及保密系统。

379

也就是说，为项目设计选择最合适的8051或其派生产品的最终决定主要依靠一系列标准，如片上ROM和RAM的数量和类型、速度以及先进的I/O端口、网络和保密要求等。

习题

14.1 调研8051派生产品的制造厂商，列出带有片上ADC或DAC的派生产品。

14.2 调研8051制造厂商，列出其他保密微控制器及它们的技术特征。

14.3 这一章中是否还有未讨论到的其他增强型8051产品？列出没有讨论到的增强功能并举出派生产品的例子。

380

① 飞利浦半导体，地址：811 E. Arques Avenue, Box 3409, Sunnyvale, CA 94088。

② Standard Microsystems公司，地址：80 Arkay Drive, Hauppauge, NY 11788。

The 8051 Microcontroller
Fourth Edition

8051微控制器（第4版）

“8051嵌入式系统开发人员的必读之作……有了它，其他同类图书都不需要了。”
——Amazon.com 读者评论

本书是讲述8051微控制器的经典著作，在工程大背景下，全面而深入地介绍了MCS-51系列微控制器的硬件体系结构和软件程序设计。初版以来被世界各大高校广泛用作教材，享有盛誉。

与同类书籍相比，本书更注重技术细节，对诸如时序和控制逻辑方面的技术细节都阐述得透彻明了；同时具有很强的工程观念，介绍了许多工程实践中必需的知识，如编程风格、需求、开发流程、系统整合和验证等。本书作者还善用举例说明的方法来论述问题，在讲解重要的基本概念和方法时都给出了恰当的例题，并在解答问题之后展开富于启发性的讨论。书中还提供了丰富而且贴近实战的接口实例和学生实习项目。通过阅读本书，读者将大大提高解决实际工程问题的能力。

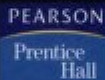
第4版新增了4章，在兼顾汇编语言的同时，全面采用8051 C语言编程，此外还新增了智能卡和数据安全等项目实例，很好地反映了最新技术趋势。

I. Scott MacKenzie 现任教于加拿大约克大学，多伦多大学博士。目前的研究兴趣是人机交互。具有近20年的一线科研和教学经验。除本书外，他还著有*Text Entry Systems* (Morgan Kaufmann, 2007) 一书。



Raphael C.-W. Phan 现任教于英国拉夫伯勒 (Loughborough) 大学，马来西亚多媒体大学博士。主要研究兴趣是密码和安全协议。任学术期刊*Cryptologia*编委。



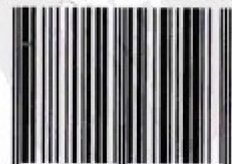


www.PearsonEd.com

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010) 88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 电子电气/嵌入式开发

人民邮电出版社网址 www.ptpress.com.cn



ISBN 978-7-115-17959-3/TN
定价：49.00 元